Automatic Space Bound Analysis for Functional Programs with Garbage Collection

Yue Niu and Jan Hoffmann
Carnegie Mellon University, Pittsburgh, PA, United States
{yuen,jhoffmann}@cs.cmu.edu

Abstract

This article introduces a novel system for deriving upper bounds on the heap-space requirements of functional programs with garbage collection. The space cost model is based on a perfect garbage collector that immediately deallocates memory cells when they become unreachable. Heap-space bounds are derived using type-based automatic amortized resource analysis (AARA), a template-based technique that efficiently reduces bound inference to linear programming. The first technical contribution of the work is a new operational cost semantics that models a perfect garbage collector. The second technical contribution is an extension of AARA to take into account automatic deallocation. A key observation is that deallocation of a perfect collector can be modeled with destructive pattern matching if data structures are used in a linear way. However, the analysis uses destructive pattern matching to accurately model deallocation even if data is shared. The soundness of the extended AARA with respect to the new cost semantics is proven in two parts via an intermediate linear cost semantics. The analysis and the cost semantics have been implemented as an extension to Resource Aware ML (RaML). An experimental evaluation shows that the system is able to derive tight symbolic heap-space bounds for common algorithms. Often the bounds are asymptotic improvements over bounds that RaML derives without taking into account garbage collection.

1 Introduction

The memory footprint of a program is an important performance metric that determines if a program can be safely executed on a given system. Ideally, developers should describe or approximate the memory footprint of programs as functions of the inputs. However, such memory bounds are often difficult to derive and to prove sound. To assist programmers with deriving memory bounds, the programming language community has developed automatic and semi-automatic analysis techniques [24, 12, 2]. These systems are often special cases of more general resource bound analyses that are based on abstract interpretation [18, 7, 36], recurrence solving [16, 1, 14, 28], type systems [27, 22, 29, 42, 41, 15], program logics [5, 10, 9, 34], proof assistants [32, 11], and term rewriting [6, 33, 17].

This article introduces a novel type system for automatically deriving upper bounds on the heap-space requirements of functional programs with garbage collection (GC). Due to the challenges of modeling and predicting garbage collection, most existing techniques for automating and guiding the derivation of bounds on the heap memory requirements assume manual memory management or simply ignore deallocation in the analysis [24, 26, 35, 13, 12, 2]. As a result, the derived bounds are not accurate when the underlying system employs garbage collection. The only exceptions we are aware of are the works by Albert et al. [3, 4], by Braberman et al. [8], and by Unnikrishnan et al. [39, 38]. They analyze the heap-space usage of programs with GC in two steps. First, they make the deallocation of GC explicit; for example with a static analysis for estimating object lifetimes [4] or with a program translation [38]. Second, they extract and solve recurrence relations to derive a bound. The difference of our work is that our technique is based on a type system, which is proved sound with respect to a formal cost semantics. Advantages of a type-based approach include natural compositionality and the use of type derivations as certificates for resource bounds.

We model the (highwater mark) memory usage based on a perfect garbage collector that immediately deallocates memory cells when they become unreachable. The bounds that are derived with this cost model are not only a good theoretical measure of the space complexity of the program but also have practical relevance. Consider a function $f : A \to B$ and assume we derived a bound $b_f : \|A\| \to \mathbb{N}$. In an execution of $f(a)$, we can then keep track of the memory usage and start the garbage collector whenever the bound
\(b_f(a)\) is reached. It is then guaranteed that the evaluation will succeed using \(b_f(a)\) heap-memory cells.\(^1\) To improve performance, we could trigger GC more often (to compactify the heap) or allow memory use of more than \(b_f(a)\) cells (to amortize the cost of garbage collection).

The first technical contribution of the work is a new operational cost semantics that models a perfect garbage collector. The cost semantics is a big-step (or natural) semantics that keeps track of the reachable memory cells in the style of Spoonhower et al. [37] and Minamide [30]. Operationally, this cost is the highwater mark on the heap usage, or the maximum number of cells used in the mutable store during evaluation. If we traverse the evaluation tree in preorder and view each node as a “step” of the computation, then a cell is used in the current node if they are reachable from the reminder of the computation. Our formalization of reachability is identical with the concept that garbage collectors implement to decide if a cell can be freed during evaluation. For simplicity, we assume that evaluation of the cons node allocates one fresh heap cell and that all other operations do not allocate heap cells. However, the semantics can be instantiated with more realistic cost metrics. A difference to existing formulations of cost semantics with GC [37, 30] is that we update the highwater mark when reachability changes at inner nodes of the derivation of the evaluation judgement instead of at leaves. Moreover, we use a freelist, which represents the cells available for evaluation. This alternative formulation is equivalent to the existing semantics and mainly motivated by the soundness proof of our type system for bound analysis. However, the cost semantics is a natural approach and different enough from its predecessors [37, 30] to be of interest in its own right.

Our second technical contribution is the type system for deriving bounds on the heap-space for programs with perfect GC. The type system is an extension of type-based automatic amortized resource analysis (AARA) [24, 27, 40, 21, 22, 31]. AARA is a template-based technique that introduces potential functions to efficiently automatically reduce bound inference to linear programming. Existing type systems based on AARA can derive bounds on the highwater mark of the heap usage for programs with manual deallocation [27], but can only derive a bound on the number number of total heap allocations for programs with GC [22]. This is usually a gross over-approximation of the actual memory requirement. Our extension is based on the observation that deallocation of a perfect collector can be modeled with destructive pattern matching (deallocate the matched cell) if data structures are used in a linear way. In the type system, we extend this observation to non-linear programs and use destructive pattern matching to accurately model deallocation even if data is shared.

The third technical contribution is to prove the soundness of the extended AARA with respect to the GC-based cost semantics. The proof is non-trivial and proceeds in two parts: First, we prove the soundness of the type system with respect to a semantics that copies data structures if they are shared. Second, we prove for all programs that our GC semantics uses less memory than this copying semantics. While the proofs are relatively standard, many details—like relating program states of the two semantics in the simulation proof—are quite involved. Briefly, we have to provide and maintain a mapping \(\gamma\) from the heap used in the GC semantics \(H_{gc}\) to subsets of the heap used in the copying semantics \(H_{copy}\) such that the image of \(H_{gc}\) under \(\gamma\) forms a partition on the second heap. The intuition is that given a cell \(l \in H_{gc}\), there must be multiple cells \(\gamma(l) \in H_{copy}\) that are were allocated during sharing, and thus are “morally the same” as \(l\).

The analysis and the cost semantics have been implemented as an extension to Resource Aware ML (RaML) [21, 22]. RaML is an implementation of AARA for a subset of OCaml that can derive multivariate polynomial bounds. However, we restrict the technical development in this paper to a simple first-order language with tuples, lists, and trees. The proofs and ideas carry over the more complex case of RaML.\(^2\) An experimental evaluation shows that the system is able to derive tight symbolic heap-space bounds for common algorithms. Our results suggest that our new analysis provides asymptotic bound improvements to several classes of commonly used functions and programming patterns. We examine the reasons for these improvements and design decisions throughout the system.

---

\(^1\)We are not considering memory fragmentation, which can be avoided using a copying collector.

\(^2\)An exception are function closures that we discuss in the Section 7.
2 Setting the Stage

In the technical part of the paper, we focus our attention to a first-order, strictly evaluated functional language. You can think this language to be a simple subset of OCaml or SML. The only recursive data type in the language is the list type. However, our work extends to the expected algebraic data types definable in RaML. Being first order, our language does not allow arbitrary local functional definitions. Instead, all functions are defined at the top level and are mutually recursive by default. The types of these functions form a signature for the program, and the semantics and typing judgments will be indexed by this signature. Thus, the function types of the language can be expressed as arrows between zero-order (base) types. Types are formally defined in Figure 1. Like in all grammars, we provide the abstract (left) and concrete (right) syntax for every type former [19]. A signature $\Sigma : \text{Var} \rightarrow \text{FTypes}$ is a map from variables to first-order types. A program $P$ is a $\Sigma$ indexed map from $\text{Var}$ to pairs $(y_f, e_f)_{f \in \Sigma}$, where $\Sigma(y_f) = \tau \rightarrow \tau'$, and $\Sigma; y_f : \tau \vdash e_f : \tau'$ (the type system is discussed in Section 4). We write $P : \Sigma$ to mean $P$ is a program with signature $\Sigma$.

To simplify the presentation, the expressions of our language (see Figure 1) are in let normal form (also A normal form). The one nonstandard construct is share $x$ as $x_1, x_2$ in $e$, which we will explain in more detail in the following sections. We introduce two distinct notions of linearity, one on the syntactic level, and one on the semantic level. Syntactic linearity is linearity in expression variables, while semantic linearity is linearity in locations (defined below). We say that a semantics is linear if it respects semantic linearity.

In line with previous works on space cost semantics [37, 30], we employ a heap, which persistently binds locations to values (normalized terms). As usual, we derive the cost of a (terminating) program from the number of heap locations used during execution, which in our case is the maximum difference between the sizes of the initial and final freelist. Locations is an infinite set of names for addressing the heap. For the rest of the paper, we use the following: $\text{Stack} \triangleq \{v \mid v : \text{Val} \rightarrow \text{Var}\}$ and $\text{Heap} \triangleq \{H \mid H : \text{Loc} \rightarrow \text{Val}\}$ for the set of stacks and heaps respectively.

Reachability Before we define the rules for the cost semantics, we relate the heap locations to values with the 3-place reachability relation $\text{reach}(H, v, L)$ on $\text{Heap} \times \text{Val} \times \wp(\text{Loc})$, where $\wp$ is the powerset. This is read as “under heap $H$, the value $v$ reaches the multiset of locations $L$”. Write $L = \text{reach}_H(v)$ to indicate this is a functional relation justified by the (valid) mode $(+, +, -)$. We also say that the reachable set of $v$ is $L$.

$A = \text{reach}_H(v_1) \quad B = \text{reach}_H(v_2) \quad A = \text{reach}_H(H(l)) \quad \{l\} \uplus A = \text{reach}_H(l) \quad v \in \mathbb{N} \cup \{T, F, \text{Null}\} \quad \emptyset = \text{reach}_H(v)$

Figure 1: Simple Types, Values, and Expressions
In the rules, \( \uplus \) is multiset union. For the rest of the paper, we will sometimes mix multiset and set operations as the situation calls for. For example, we will write \( l \in S \) for a multiset \( S \) if \( S(l) \geq 1 \). Complete definitions and notations can be found in the appendix 7.

The notion of reachability naturally lifts to expressions:

\[
\text{locs}_{V,H}(e) = \bigcup_{x \in FV(e)} \text{reach}_H(V(x))
\]

Where \( FV : \text{Exp} \to \mathcal{P}(\text{Var}) \) denotes the set of free-variables of expressions as usual.

**Towards the Garbage Collection Cost Semantics** Now we are ready to give a first attempt to modeling the cost semantics for a tracing garbage collector. Before we present our new semantics, we explain an existing cost semantics we experimented with [30]. Judgements have the form \( V, H, R \vdash e \Downarrow^s v, H', \) which can be read as follows. Under stack \( V \in \text{Stack} \), heap \( H \in \text{Heap} \), and continuation set \( R \subseteq \text{Loc} \), \( e \) evaluates to \( v \) and \( H' \) using \( s \) heap locations. The idea is that \( R \) keeps track of the set of locations necessary to complete the evaluation after \( e \) is evaluated (hence the name continuation). For example, we have the let rule:

\[
V, H, R \uplus \text{locs}_{V,H}(x,e_2) \vdash e_1 \Downarrow^{s_1} v_1, H_1 \quad V[x \mapsto v_1], H, R \vdash e_2 \Downarrow^{s_2} v_2, H_2
\]

\[
\vdash V, H, R \vdash \text{let}(e_1; x : \tau.e_2) \Downarrow^{[\max(s_1,s_2)]} v_2, H_2
\]

Notice that to evaluate \( e_1 \), we have to extend the continuation \( R \) with locations in \( e_2 \), which will be used after \( e_1 \) is evaluated. The total space used is the max of the component, indicating that locations used for \( e_1 \) can be reused for \( e_2 \). This is clear when we look at the variable rule.

\[
V(x) = v
\]

\[
V, H, R \vdash x \Downarrow^{\text{dom(\text{reach}_H(v))}} v, H
\]

It states that evaluating a variable \( x \) requires the locations reachable from \( x \) as well as the continuation set \( R \). While this way of counting heap locations does model a tracing garbage collector, it is not compatible with the existing type systems for amortized analysis. In these systems, the type rules count the heap locations as data is created, i.e. at each data constructor. Thus looking up a variable incurs no cost, since it was accounted for during creation. This mismatch between the dynamics and statics of language prevents us from proving the soundness of the analysis.

### 3 Garbage Collection Cost Semantics

In this section, we present our novel cost semantics by combining freelist semantics from [25] with the cost semantics for modeling perfect GC [30] that we discussed in the previous section. The resulting semantics, called \( E_{gc} \), is well suited for proving the soundness of the novel type-based bound analysis.

The garbage collection cost semantics \( E_{gc} \) is defined by a collection of judgement of the form

\[
\mathcal{C} \vdash_{\Sigma} e \Downarrow v, H', F'
\]

Where \( \mathcal{C} \in \text{Stack} \times \text{Heap} \times \varphi(\text{Loc}) \times \mathcal{P}(\text{Loc}) \) is a configuration usually written with variables \( V, H, R, F \). Because the signature \( \Sigma \) for the mapping of function names to first-order functions does not change during evaluation, we drop the subscript \( \Sigma \) from \( \vdash_{\Sigma} \) when the context of evaluation is clear. Given a configuration \( \mathcal{C} = (V, H, R, F) \), the evaluation judgment states that under stack \( V \), heap \( H \), continuation (multi)set \( R \), freelist \( F \), and program \( P \) with signature \( \Sigma \), the expression \( e \) evaluates to value \( v \), and engenders a new heap \( H' \) and freelist \( F' \). In comparison with the attempt from the previous section, the key ingredient we added is the freelist, which serves as the set of available locations. We call \( R \) a (multi)set since the fact that it’s a multiset is only useful during the soundness proof. For evaluation, it is convenient to just view \( R \) as a set; the same is true for \( \text{reach}_H(v) \).
The semantics $\mathcal{E}_{gc}$ is designed to model the heap usage of a program running with a tracing counting garbage collector: whenever a heap cell becomes unreachable from the root set, it becomes collected and added to the freelist as available for reallocation. As before, the continuation set $\mathcal{R}$ represents the set of locations required to compute the continuation excluding the current expression. We define the root set as the union of the locations in the continuation set $\mathcal{R}$ and the locations in the current expression $e$.

The inference rules for the semantics are given in Figure 2. For example, the rule F:CondT states that, to evaluate a conditional, look in the stack for the value of the branching boolean. In the case it is true, we evaluate the first expression with the corresponding restricted stack $V_1$ and the original continuation set $\mathcal{R}$.

Another example is the rule F:Let for let expressions: to evaluate the expressions $\text{let}(e_1; x: \tau, e_2)$, we evaluate the first expression with the corresponding restricted stack $V_1$ and a expanded continuation set $\mathcal{R}'$. The extra locations come from the free variables of $e_2$ (not including the bound variable $x$), which we cannot collect during the evaluation of $e_1$. Next, we extend the restricted stack $V_2$ with the result $v_1$, and evaluate $e_2$ with this stack and the original continuation set $\mathcal{R}$. The other rules are similar.

Note that in contrast to the semantics in the previous section, evaluating a variable does not incur any cost. This ensures that we will be able prove the soundness of the type system. Also, since we don’t
allow local function definitions, we do not create closures during evaluation. Also note that we restrict the domain of the stack to the appropriate variables during evaluation. This is only to facilitate the proof of the linearity of the copying semantics introduced later, and not necessary for the implementation. These issues are discussed in 6.

For example, we can implement the `append` and `appTwice` function, which has variable sharing. First, we analyze the heap usage of `append` under \( \mathcal{E}_{\text{gc}} \). We case on the first component of the input. In case it’s \( \text{nil} \), we just return \( 12 \), and there are no allocations or deallocations. In case it’s cons of \( x \) and \( xs \), we need to allocate one heap location for the cons cell binding \( x \) and the recursive result, for which we can use the just matched-on cell. Again, the total space overhead of `append` is zero. For `appTwice`, we first share the list \( 1 \) as \( 11 \) and \( 12 \). In the first `let`, the locations in \( 12 \) are added to the continuation set, which prevents the first call to `append` from destroying \( 11 \). Thus size of \( 11 \) new locations are allocated from the freelist to construct \( 11' \). The second call has no net increase in heap allocations since \( 12 \) can be destructed along the way. The return value is a pair which is stack-allocated and doesn’t require a heap allocation. Thus, the total space overhead for `appTwice` is size of input list \( 1 \).

From this, we see that the minimum size for the initial freelist to successively evaluate a call to `appTwice` is exactly the length of the input. In general, we define the cost of a closed program to be the minimum size of the initial freelist that guarantees successful evaluation. Although we don’t prove it in this paper, this can be seen to be equivalent to the cost annotation in previous cost semantics introduced in section 2, which justifies \( \mathcal{E}_{\text{gc}} \) as a cost semantics.

4 Automatic Amortized Heap-Space Analysis with GC

Automatic Amortized Resource Analysis (AARA) The idea of AARA [24, 27, 21, 22] is to automate the potential method of amortized analysis using a type system. Types introduce potential function that map data structures of the given type to non-negative numbers. The type rules ensure that there is always sufficient potential to cover the evaluation cost of the next step and the potential of the next program state.

To illustrate the idea, we informally explain the linear potential method for the functions in Figure 3. We will use the allocation/heap metric which simply counts the number of cons constructor calls during the evaluation.\(^3\) With this metric, the cost of evaluating `append(11, 12)` is \( m \), where \( m \) is the number of cons constructors in \( 11 \), and the resource annotated type of append is \( L^1(\text{int}) \times L^0(\text{int}) \xrightarrow{0/0} L^1(\text{int}) \). This type says that to type `append(11, 12)`, we need \( 11 \) to have 1 potential per element, \( 12 \) to have 0 potential per element, and the result will be a list with 0 potential per element. Additionally, the function uses 0 constant potential, and leaves 0 constant potential after evaluating. This translates to a bound which states that the number of allocations `append` makes is bounded by 1 times size of the first list. For `appTwice(1)`, the cost under the heap metric is \( 2m \), where \( m \) is the number of cons constructors in \( 1 \). This is because we have to share the input list across two calls of `append`, which each requires lists with unit potential per element. For example, if \( 1 : L^2(\text{int}) \), then \( 11 \) and \( 12 \) both get 1 potential per element so that \( 11 : L^1(\text{int}) \), \( 12 : L^1(\text{int}) \), which covers the cost of the next 2 calls to `append`, and the resulting pair of lists both have 0 potential per element. More generally, we can give the following types to `append` and `appTwice`:

\[
\text{append} : L^p(\text{int}) \times L^q(\text{int}) \xrightarrow{r/s} L^s(\text{int}) \text{ such that } p \geq s + 1 \text{ and } q \geq s
\]
\[
\text{appTwice} : L^p(\text{int}) \xrightarrow{q/r} L^r(\text{int}) \times L^s(\text{int}) \text{ such that } p \geq r + s + 2
\]

\(^3\)This is in contrast to the highwater mark for the GC semantics \( \mathcal{E}_{\text{gc}} \) that is targeted by our new analysis.
A ∈ BType
\[ H \models \text{val}(\text{Null}) \rightarrow \text{val}(\text{Null}) : L(A) \] (V:Nil)
\[ H \models \text{val}(\text{Null}) \rightarrow \text{val}(\text{Null}) : \text{unit} \] (V:Const)
\[ H \models \text{val}(\text{Null}) \rightarrow \text{val}(\text{Null}) : \text{unit} \] (V:ConstI)
\[ A \rightarrow a \] (V:Nil)
\[ A \rightarrow a \] (V:Const)
\[ H \models \text{val}(\text{Null}) \rightarrow \text{val}(\text{Null}) : \text{unit} \] (V:ConstI)
\[ A \rightarrow a \] (V:Nil)
\[ A \rightarrow a \] (V:Const)
\[ H \models \text{val}(\text{Null}) \rightarrow \text{val}(\text{Null}) : \text{unit} \] (V:ConstI)
\[ A \rightarrow a \] (V:Nil)
\[ A \rightarrow a \] (V:Const)
\[ H \models \text{val}(\text{Null}) \rightarrow \text{val}(\text{Null}) : \text{unit} \] (V:ConstI)
\[ A \rightarrow a \] (V:Nil)
\[ A \rightarrow a \] (V:Const)
\[ H \models \text{val}(\text{Null}) \rightarrow \text{val}(\text{Null}) : \text{unit} \] (V:ConstI)
\[ A \rightarrow a \] (V:Nil)
\[ A \rightarrow a \] (V:Const)
\[ H \models \text{val}(\text{Null}) \rightarrow \text{val}(\text{Null}) : \text{unit} \] (V:ConstI)
\[ A \rightarrow a \] (V:Nil)
\[ A \rightarrow a \] (V:Const)
\[ H \models \text{val}(\text{Null}) \rightarrow \text{val}(\text{Null}) : \text{unit} \] (V:ConstI)
\[ A \rightarrow a \] (V:Nil)
\[ A \rightarrow a \] (V:Const)
\[ H \models \text{val}(\text{Null}) \rightarrow \text{val}(\text{Null}) : \text{unit} \] (V:ConstI)
\[ A \rightarrow a \] (V:Nil)
\[ A \rightarrow a \] (V:Const)
\[ H \models \text{val}(\text{Null}) \rightarrow \text{val}(\text{Null}) : \text{unit} \] (V:ConstI)
\[ A \rightarrow a \] (V:Nil)
\[ A \rightarrow a \] (V:Const)
\[ H \models \text{val}(\text{Null}) \rightarrow \text{val}(\text{Null}) : \text{unit} \] (V:ConstI)
\[ A \rightarrow a \] (V:Nil)
\[ A \rightarrow a \] (V:Const)
\[ H \models \text{val}(\text{Null}) \rightarrow \text{val}(\text{Null}) : \text{unit} \] (V:ConstI)
\[ A \rightarrow a \] (V:Nil)
\[ A \rightarrow a \] (V:Const)
\[ H \models \text{val}(\text{Null}) \rightarrow \text{val}(\text{Null}) : \text{unit} \] (V:ConstI)
\[ A \rightarrow a \] (V:Nil)
\[ A \rightarrow a \] (V:Const)
\[ H \models \text{val}(\text{Null}) \rightarrow \text{val}(\text{Null}) : \text{unit} \] (V:ConstI)
\[ A \rightarrow a \] (V:Nil)
\[ A \rightarrow a \] (V:Const)
\[ H \models \text{val}(\text{Null}) \rightarrow \text{val}(\text{Null}) : \text{unit} \] (V:ConstI)
\[ A \rightarrow a \] (V:Nil)
\[ A \rightarrow a \] (V:Const)
\[ H \models \text{val}(\text{Null}) \rightarrow \text{val}(\text{Null}) : \text{unit} \] (V:ConstI)
\[ A \rightarrow a \] (V:Nil)
\[ A \rightarrow a \] (V:Const)
\[ \Sigma; x : B \frac{p}{q} \rightarrow x : B \]

(L:Var) \[ \Sigma(f) = A \frac{q}{q'} \rightarrow B \]

\[ \Sigma; x : A \frac{q}{q'} f(x) : B \]

(Sigma) \[ \Sigma; \Gamma = \frac{\Sigma; \Gamma}{q} e_1 : B \]

\[ \Sigma; \Gamma, x : bool \frac{q}{q'} \rightarrow \begin{cases} e_1 : B & \text{if } x \text{ then } e_1 \text{ else } e_f : B \end{cases} \]

(L:Cond)

\[ \Sigma; x_1 : A_1, x_2 : A_2 \frac{q}{q'} (x_1, x_2) : A_1 \times A_2 \]

(L:Pair)

\[ \Sigma; x : (A_1, A_2) \frac{q}{q'} \rightarrow \begin{cases} e_1 \rightarrow B \end{cases} \]

(Sigma) \[ \Sigma; \Gamma, x : \begin{cases} e_1 : B \end{cases} \]

\[ \Sigma; \Gamma, x : (A_1, A_2) \frac{q}{q'} \rightarrow \begin{cases} e_1 \rightarrow B \end{cases} \]

(L:MatP)

\[ \Sigma; 0 \frac{q}{q'} \rightarrow \begin{cases} nil : L^p(A) \end{cases} \]

(L:Nil)

\[ \Sigma; \Gamma; x_1 : A_1, x_2 : A_2 \frac{q}{q'} \rightarrow \begin{cases} e \rightarrow B \end{cases} \]

(Sigma) \[ \Sigma; \Gamma, x : L^p(A) \frac{q}{q'} \rightarrow \begin{cases} cons(x_1; x_2) : L^p(A) \end{cases} \]

(L:Cons)

\[ \Sigma; \Gamma; x : L^p(A) \frac{q}{q'} \rightarrow \begin{cases} match \ x \begin{cases} \text{nil} \leftarrow e_1 \end{cases} \end{cases} \]

(L:MatL)

\[ \Sigma; \Gamma; x : A \frac{q}{q'} \rightarrow \begin{cases} share \ x \text{ as } x_1, x_2 \end{cases} \]

(L:Share)

\[ \Sigma; \Gamma; x : A \frac{q}{q'} \rightarrow \begin{cases} let(e_1 : x \tau, e_2) : B \end{cases} \]

(L:Let)

\[ \Phi_A([a_1, \ldots, a_n] : L^p(A)) = p \cdot n + \sum_{1 \leq i \leq n} \Phi_A(a_i : A) \]

Now define \( A \gamma_A, A_2, n \) as the sharing relation for resource-annotated types:

\[ L^p(A) \gamma^n L^p(A_1), L^p(A_2) \]

if \( p = q + r + n \) and \( A \gamma^n A_1, A_2 \)

\[ A \times B \gamma^n A_1 \times B_1, A_2 \times B_2 \]

if \( A \gamma^n A_1, A_2 \) and \( B \gamma^n B_1, B_2 \)

\[ A \gamma^n A, A \]

if \( A \in \{ \text{unit, bool, nat} \} \)

The sharing relation captures the amount of potential needed to copy a type \( A \) where each cons node in any structure in \( [A] \) has a copying overhead \( n \).

**Type Rules** The type system \( \text{FO}^{\text{pc}} \) consists of rules of the form \( \Sigma; \Gamma : e : A \), read as under signature \( \Sigma : \text{Var} \rightarrow \text{FTypes} \), context \( \Gamma : \text{Var} \rightarrow \text{BTypes} \), \( e \) has type \( A \) starting with \( q \) units of constant potential and ending with \( q' \) units.

Our type system is based on the one of classic linear AARA \cite{24}. We give a review of the rules in Figure 5. Since we are interested in the number of heap locations, there is an implicit side condition in all rules which ensures all constants are assumed to be nonnegative.

For example, \( \text{L:Cons} \) states that to add an element to a list with \( p \) potential per element, we need \( p + 1 \) units of constant potential: \( p \) to maintain the potential of the list, and 1 for allocating the cons cell. \( \text{L:MatL} \) states that matching on a list with type \( L^p(A) \), we need to type the nil case with the same constant potentials, and we need to type the cons case with an additional \( p \) units of constant potential, since we get the spill of \( p \) from the definition of linear potential. As the last example, we look at \( \text{L:Share} \), which states that to share a variable \( x \) of type \( A \), we need to split the potential between \( A_1 \) and \( A_2 \), and type the rest of the expression with the two new variables \( x_1 : A_1, x_2 : A_2 \).

**New Rules** The new type system for programs with garbage collection replaces the rules \( \text{L:MatL} \) and \( \text{L:Share} \). The observation is that if we ensure that locations are used linearly, we can use destructive pattern matching to model local garbage collection by returning the potential associated with the constructor location (notice the extra \(+1\) in the second premise):

\[ \Sigma; \Gamma, x : L^p(A) \frac{q}{q'} \rightarrow \begin{cases} match \ x \begin{cases} \text{nil} \leftarrow e_1 \end{cases} \end{cases} \]

\[ \Sigma; \Gamma, x : L^p(A) \frac{q}{q'} \rightarrow \begin{cases} cons(x_1; x_2) \leftarrow e_1 \end{cases} \]

\( (\text{L:MatLD}) \)
This is validated by the fact (Lemma 5) that in the auxiliary copying semantics (introduced in Section ??), once a cons-cell is matched on, there can be no live references from the root set to it, and thus we are justified in restituting the potential to type the subexpression \( e_2 \).

However, the rule L:MatLD is not sound for programs with aliasing of data. We address this issue by replacing the rule L:Share with the rule L:ShareCopy:

\[
\frac{\Sigma; \Gamma, x_1 : A_1, x_2 : A_2 \vdash^q e : B}{\Sigma; \Gamma, x : A \vdash^q \text{share } x \text{ as } x_1, x_2 \text{ in } e : B} \quad \text{(L:ShareCopy)}
\]

To share a variable of type \( A \), we need to split the potential between two new annotated types \( A_1 \) and \( A_2 \) as usual. In addition, we have to pay an “overhead” of 1 for every cons node in any structure in \([A]\). The idea is that we treat data as if it is actually copied. This is sound w.r.t the copying semantics because the size of the domain of the reachable set of a value \( v \) is exactly the linear potential of \( v : A \) with all resource annotations set to 1.

Now, we can give now the improved space overhead bound to append and appTwice:

\[
\begin{align*}
\text{append} & : L^p(\text{int}) \times L^q(\text{int}) \rightarrow L^r(\text{int}), \quad \text{where } p \geq s \text{ and } q \geq s \\
\text{appTwice} & : L^p(\text{int}) \rightarrow L^r(\text{int}) \times L^r(\text{int}), \quad \text{where } p + q \geq r + s + 1
\end{align*}
\]

**Cost Metrics** In previous versions of AARA [27, 21], the typing judgment and cost semantics are parametrized by a cost metric \( m : \text{res} \cdot \text{const} \rightarrow \mathbb{Q} \), which assigns a constant cost to each step in the semantics. Since the cost for all program construct is zero save for the cons data constructor, we elide the cost metric altogether. Although we defined the constructor to cost 1 heap location (as shown in L:Cons and L:MatLD), it can be any constant as long as the introduction and elimination rules agree on that constant. Thus we can extend the type system to accurately track closure sizes and constructor which vary in size depending on the argument (more in Section 7).

**Type Inference** One of the benefits of AARA is efficient type inference using off-the-shelf LP solvers [24], even for non-linear potential functions [21, 22]. The new rules do not complicate inference and previous techniques still apply. In a nutshell, inference is performed in three steps: First, perform a standard Hindley-Milner type inference for the base types. Then, annotate the type derivation with (yet unknown) variables for the potential annotations and collect linear constraints that are derived from the type rules. Finally, solve the constraints with an LP solver and minimize the potential annotations of the inputs. Details can be found in previous work [24, 22].

## 5 Soundness of \( \text{FO}^{gc} \)

We seek to prove the following theorem.

**Theorem 1** (Soundness). Let \( H \vdash V : \Gamma \), \( \Sigma; \Gamma \vdash^q e : B \), and \( V, H \vdash e \downarrow v, H' \). Then for all configurations \( W, Y, F, R \): If \( V, H \sim W, Y \) and \( |F| \geq \Phi_{V, H}(\Gamma) + q \), then there exists a value \( w \), and a freelist \( F' \) such that

\[
W, Y, R, F \vdash \downarrow e \downarrow w, Y', F' \quad \text{ and } \quad v \sim^{t_i}_{t_j} w.
\]

Here, \( V, H \vdash e \downarrow v, H' \) is a standard big-step evaluation judgment derived from \( \mathcal{E}_{gc} \). \( V, H \sim W, Y \) is context equivalence, and \( v \sim^{t_i}_{t_j} w \) is value equivalence (these are defined below). The theorem states that, given a terminating expression and a freelist that is sufficiently large (as predicated by the type derivation), a run with \( \mathcal{E}_{gc} \) will normalize to an equivalent value.

To facilitate the proof, we define an intermediate semantics \( \mathcal{E}_{copy} \) which is semantically linear. The proof has two stages: First, we show \( \mathcal{E}_{copy} \) over-approximates \( \mathcal{E}_{gc} \), meaning that any computation that succeeds with \( \mathcal{E}_{gc} \) will succeed with an equally-sized or smaller freelist with \( \mathcal{E}_{gc} \). Then we show \( \text{FO}^{gc} \) is sound with respect to \( \mathcal{E}_{copy} \), and thus by the previous step sound with respect to \( \mathcal{E}_{gc} \).
Definition 1 (Value Equivalence). Two values \( v_1, v_2 \) are equivalent (with the presupposition that they are well-formed w.r.t heaps \( H_1, H_2 \)), iff \( H_1 \models v_1 \Rightarrow a : A \) and \( H_2 \models v_2 \Rightarrow a : A \). Write value equivalence as \( v_1 \sim^{H_1}_{H_2} v_2 \).

Definition 2 (Context Equivalence). Two contexts \( (V_1, H_1), (V_2, H_2) \) are equivalent (with the presupposition that both are well-formed contexts) iff \( \text{dom}(V_1) = \text{dom}(V_2) \) and for all \( x \in \text{dom}(V_1) \), \( V_1(x) \sim^{H_1}_{H_2} V_2(x) \). Write context equivalence as \( (V_2, H_2) \sim (V_2, H_2) \).

Stated simply, two contexts are equivalent when they have the same domain and equal semantic values (structures defined in Section ??).

Linear Garbage Collection Cost Semantics To establish the soundness of the type system, we need an intermediary semantics \( \mathcal{E}_{\text{copy}} \) which is semantically linear. As mentioned in Section 2, this means that locations are linear, that is, no location can be used twice in a program. Variable sharing is achieved via copying: the shared value is created by allocating a fresh set of locations from the freelist and copying the locations of the original value one by one. This is also sometimes referred to as deep copying. Let \( \text{copy}(H, L, v, H', v') \) be a 5-place relation on \( \text{Heap} \times \mathcal{P}(\text{Loc}) \times \text{Val} \times \text{Heap} \times \text{Val} \). Similar to reachability, we write this as \( H', v = \text{copy}(H, L, v) \) to signify the intended mode for this predicate: \((+, +, +, +, –, –)\).

\[
\begin{align*}
v \in \{n, T, F, \text{Null}\} & \quad \quad l' \in L & \quad \quad H', v = \text{copy}(H, L, v) \\
H, v = \text{copy}(H, L, v) & \quad \quad H' = \text{copy}(H, L, l) & \quad \quad \text{dom}(\text{reach}_H(v_1)) = |L_1| & \quad \quad H_1, v_1 = \text{copy}(H, L, v_1) \\
& \quad \quad H_2, v_2 = \text{copy}(H, L, v_2) & \quad \quad H_2, \langle v_1, v_2 \rangle = \text{copy}(H, L, \langle v_1, v_2 \rangle)
\end{align*}
\]

Primitives require no cells to copy; a location value is copied recursively; a pair of values is copied sequentially, and the total number of cells required is the size of the reachable set of the value. Now, consider \( \mathcal{E}_{\text{gc}} \) with the share rule \( \text{F:Share} \) replaced with the following rule.

\[
\begin{align*}
L \subseteq F & \quad \quad |L| = |\text{dom}(\text{reach}_H(v'))| & \quad \quad H', v'' = \text{copy}(H, L, v') \\
& \quad \quad V(x) = v & \quad \quad H_2 = (V[x_1 \rightarrow v', x_2 \rightarrow v'']) & \quad \quad |F'v(e)| \\
F' = F \setminus L & \quad \quad g = \{ l \in H \mid l \notin F' \cup R \cup \text{locsv}_{L_2}(e) \} & \quad \quad V_2, H', R, F' \cup g \vdash e \Downarrow v, H'', F'' \\
& \quad \quad V, H, R, F \vdash \text{share} x \text{ as } x_1, x_2 \text{ in } e \Downarrow v, H'', F'' & \quad \quad \text{(E:Share)}
\end{align*}
\]

To share a variable, we first copy the shared value. The number of cells required is equal to the size of the reachable set from the value. This copying sharing semantics is what justifies the analysis to use restitute the potential when matching on a cons node, since even if the node was shared, we had to pay for the cost by copying the node when sharing the original value. Next, we restrict the stack to the appropriate variables. Lastly, any locations not reachable from the current subexpression \( e \) is collected. This is for the case when a variable is shared but not used later.

Intuitively, we expect that any terminating computation with \( \mathcal{E}_{\text{copy}} \) has a corresponding run with \( \mathcal{E}_{\text{gc}} \) that can be instantiated with an equally-sized or smaller freelist. Although this seems quite straightforward to prove, a complete characterization of the relationship between the space allocations of two runs with each semantics is necessary. To motivate this, consider the following proof attempt:

**Attempt 1.** Let \( \mathcal{C}_e = (V, H, R, F) \) be a configuration and \( (\mathcal{C}_e, e) \) be a linear computation. Given that \( \mathcal{C}_e \vdash \text{free} e \Downarrow w, Y', M' \) and \( H' \Downarrow v \Rightarrow a : A \), for all configurations \( \mathcal{C}_1 = (W, Y, R, M) \) such that \( W, Y \sim V, H \) and \( |M| = |F| \), there exists a triple \( (w, Y', M') \in \text{Val} \times \text{Heap} \times \text{Loc} \) such that \( \mathcal{C}_1 \vdash \text{free} e \Downarrow w, Y', M' \) and \( v \sim^{H' \Downarrow Y \Downarrow} w \) and \( |M'| \geq |F'| \).

We proceed with induction on the derivation of the judgment in \( \mathcal{E}_{\text{copy}} \). Almost every case goes through, save for E:Let. First, we get \( W_1, Y \sim V_1, H \) and We have the following from induction on the first premise:

\[
W_1, Y, R', M \vdash \text{free} e \Downarrow w_1, Y_1, M_1 \quad \quad v_1 \sim^{H_1 \Downarrow Y_1 \Downarrow} w_1 \quad \quad |M_1| \geq |F_1|
\]
To instantiate the induction hypothesis on the second premise, we need to show that, among other things, 
\(|M_1 \cup j| \geq |F_1 \cup g|\), where \(j\) is the set of collected locations in the \(E_{gc}\) judgment. We cannot show this
precisely because \(g\) might contain more cells then \(j\) due to the linearity of \(E_{copy}\), thus preventing a piecewise comparison. But of course \(|j|\) is always less than \(|g|\), since \(E_{gc}\) doesn’t copy to share values! This shows
that there is a mismatch between the induction hypothesis and the relationship between the sizes of the
respective freelists and the garbage sets.

Before giving the full relationship between two configurations that allows the successful proof, We first
give coherence conditions to a configuration. For a configuration \((V, H, R, F)\), denote the garbage w.r.t a set
of locations \(L\) as collect\((R, L, H', F')\) = \(\{l \in H' \mid l \notin F' \cup R \cup L\}\).

**Definition 3.** A configuration \((V, H, R, F)\) is well-formed if
1. \(\text{dom}(H) \subseteq \text{reach}_H(V) \cup R \cup F\)
2. \(\text{reach}_H(V) \cup R \subseteq \text{dom}(H) \setminus F\)
3. \(\text{collect}(R, \text{reach}_H(V), H, F) = \emptyset\)

The well-formed conditions ensure the stack and continuation sets are within the active region of the
heap \(H \setminus F\), and that the active region of the heap does not contain garbage – all garbage locations are
already in the freelist. Although we can leave this condition out, it unnecessarily complicates the presentation
of the theorem.

Now, we can present the following criteria which characterizes the required equivalence between two
configuration, called copy extension.

**Definition 4.** A well-formed configuration \(C_2 = (V_2, H_2, R_2, F_2)\) is a copy extension of another well-formed
configuration \(C_1 = (V_1, H_1, R_1, F_1)\) iff
1. \(V_1, H_1 \sim V_2, H_2\)
2. There is a proper partition \(\gamma\): \(\text{dom}(H_1) \setminus F_1 \to \mathcal{P}(\text{dom}(H_2) \setminus F_2)\) such that for all \(l \in \text{dom}(\gamma)\),
   \(|\gamma(l)| = \text{reach}_{H_1}(V_1)(l) + R_1(l)\)
3. For all \(l \in \text{dom}(\gamma), x \in \text{dom}(V_1), \text{valid sequence of directions } P\text{ w.r.t } V_1(x),
   |\text{reach}_{H_2}(V_2(x; P)) \cap \gamma(l)| = \text{reach}_{H_1}(V_1(x; P))(l)\).
4. For all \(l \in \text{dom}(\gamma), |\gamma(l) \cap R_2| = R_1(l)\)
5. \(|F_1| = |F_2| + |\ominus(\gamma)|\), where \(\ominus(\gamma) = \bigcup_{P \in \text{ec}(\gamma)} P \setminus (\text{rep}(P))\)

Write this as \(C_1 \preceq C_2\).

The intention is that \(C_2\) is a configuration for initiating an evaluation using \(E_{copy}\), and \(C_1\) a configuration
for \(E_{gc}\).

The first condition is the straightforward context equivalence. The second condition requires the existence
of a mapping \(\gamma\) that tells us given a location in \(H_1\), which locations in \(H_2\) are shared instances.

For example, consider the expression \texttt{share x as x1, x2 in e} and assume the stack is \([x \mapsto 1]\), and
the heap equals \([1 \mapsto \{0,\text{Null}\}], i.e. \(x\) is the list \([0]\). In an evaluation with \(E_{gc}\), the stack becomes
\([x1 \mapsto 1, x2 \mapsto 1]\), and the heap does not change. With \(E_{copy}\), we allocate a new location in the heap:
\([1 \mapsto \{0,\text{Null}\}, 2 \mapsto \{0,\text{Null}\}]\), and the stack changes accordingly: \([x1 \mapsto 1, x2 \mapsto 2]\). Now \(\gamma\) would map 1 to
\([1, 2]\), since both are shared instances of the former.

Thus, each \(\gamma(l)\) is mapped to a disjoint subset in \(H_2\), and each location in \(H_2\) would have a representative
in \(H_1\). Furthermore, we noticed it is crucial to include the fact that the size of \(\gamma(l)\) must be the sum of the
number of references from the stack and the continuation set. Furthermore, we also require each subset \(\gamma(l)\)
(also referred to as class) to be nonempty (this is the proper partition condition).

While \(\gamma\) gives us a relation between the two respective heaps, we still need to know exactly how variables
on the stack factor in this relationship. Let \(l \in H_1\). Specifically, we need to know that the number of
references to \(l\) from every subvalue in \(V_1\) is equal to the size of the corresponding part of the class \(\gamma(l)\). First,
we need to access subvalues of a value using directions:
Definition 5. Let \( \text{dir} \) be the set \{L, R, N\}, denoting left, right, and next respectively. We can index values via directions:

\[
\begin{align*}
\text{get}_H(\text{Just}((v_1, v_2), L)) &= \text{Just}(v_1) & \text{get}_H(\text{Just}((v_1, v_2), R)) &= \text{Just}(v_2) \\
\text{get}_H(\text{Just}((v_1, v_2), \gamma)) &= \text{None} & \text{get}_H(\text{Just}(l), N) &= \text{Just}(H(l)) \\
\text{get}_H(\text{Just}(l), \gamma) &= \text{None} & \text{get}_H(r, \gamma) &= r
\end{align*}
\]

Let \( P \) be a sequence of directions. Extend \( \text{get} \) to sequence of directions:

\[
\begin{align*}
\text{find}_H(v, D :: P) &= \text{find}_H(\text{get}_H(v, D), P) \\
\text{find}_H(v, []) &= v
\end{align*}
\]

Call \( P \) valid w.r.t a value \( v \) if \( \text{find}_H(v, P) = \text{Just}(v') \) for some \( v' \). Write \( V_H(x; P) \) for \text{fromJust}(\text{find}_H(V(x), P)) given a valid sequence \( P \) w.r.t \( V(x) \), and \( \text{reach}_H(V(x; P)) \) for \text{reach}_H(V_H(x; P)).

With this, the third condition gives us a more finned grained restriction: for any subvalue in \( V_1 \), the number of references from it to \( l \) is equal to the size of the intersection of the reachable set of the corresponding subvalue in \( V_2 \) with the appropriate class \( \gamma(l) \).

The next condition simply states that the continuation sets respect \( \gamma \). Lastly, we have have that \( F_1 \) is greater \( F_2 \), with the overhead \( \circ(\gamma) \) being exactly the sum \( \sum_{l \in \gamma} |\gamma(l)| - 1 \) (since each class is non-empty, we use \( \text{rep}(l) \) to choose an arbitrary element of the class \( \gamma(l) \)).

Now we can formulate the key lemma:

Lemma 2. Let \((C_2, e)\) be a linear computation. Given that \( C_2 \vdash \text{copy} e \vdash v, H', F' \), and \( H' = v \rightarrow a : A \), for all well-formed configurations \( C_1 \) such that \( C_1 \preceq C_2 \), there exists a triple \((w, Y', M') \in \text{Val} \times \text{Heap} \times \text{Loc}\) and \( \gamma' : \text{dom}(Y') \rightarrow \mathcal{P}(\text{dom}(H') \setminus F') \) s.t.

1. \( C_1 \vdash \text{free} e \vdash w, Y', M' \)
2. \( v \sim_{Y'} w \)
3. \( \gamma' \) is a proper partition, such that for all \( l \in \text{dom}(\gamma') \), \( |\gamma'(l)| = |\text{reach}_{Y'_1}(w_1)(l)| + S(l) \)
4. \( \text{For all } P, |\text{reach}_H(\text{find}_H(v; P)) \cap \gamma'(l)| = \text{reach}_{Y'_1}(\text{find}_{Y'_1}(w; P))(l) \)
5. \( \text{For all } l \in \text{dom}(\gamma'), \gamma'(l) \cap R = \gamma(l) \cap R \)
6. \( |M'| = |F'| + |\circ(\gamma')| \)

The third condition deserves some explanation. The size of \( \gamma(l) \) should equal the total number of ways \( l \) could be reached from the root set in the \( \mathcal{E}_g \) evaluation. This includes \( \text{reach}_{Y'_1}(w_1)(l) \) and \( S(l) \), but also any paths to \( l \) had been collected during the run (this accounts for unused variables that referenced \( l \)).

Thus, we have shown that we can execute a computation using the \( \mathcal{E}_g \) if the computation succeeded in a run with \( \mathcal{E}_{\text{copy}} \), and that indeed \( \mathcal{E}_{\text{copy}} \) is an over approximation of \( \mathcal{E}_g \).

**Soundness of \( \mathcal{E}_{\text{copy}} \)** As mentioned above, we introduce an standard big step semantics \( V, H \vdash e \downarrow v, H' \) that does not use freelist or accounts for garbage collection. We use it to characterize expressions that normalize to values when initialized with a sufficient freelist. This technique has also be employed in earlier work on AARA [25].

Let this auxiliary semantics be \( \mathcal{E}_{\text{oper}} \). Here, the “freelist” is the whole ambient set of locations \( \text{Loc} \), thus we never run out of locations during evaluation. This introduces a problem already present in the previous section when we related \( \mathcal{E}_{\text{copy}} \) and \( \mathcal{E}_g \), which we solved with value and context equivalence. We give a bit more explanation here. When comparing evaluation results between a run with \( \mathcal{E}_{\text{copy}} \) and \( \mathcal{E}_{\text{oper}} \), as the return values might not be syntactically equal. Consider the following expression \( \text{let} \ a = [4] \) in \( [5] \) Assuming locations are natural numbers, and we run \( \mathcal{E}_{\text{copy}} \) with the freelist \{1\}. First, 1 is allocated and mapped to \([4]\). Then, since the first subexpression \([4]\) is not used afterwards, we collect 1, and reuse it and map again to \([5]\). Thus the return value is 1. In a run with \( \mathcal{E}_{\text{oper}} \), we also first map 1 to \([4]\), but then allocate a new
Definition 6. (Linear context) Given a context \((V, H)\), let \(x, y \in \text{dom}(V)\), \(x \neq y\), and \(r_x = \text{reach}_H(V(x))\), \(r_y = \text{reach}_H(V(y))\). It is linear given that \(\text{set}(r_x), \text{set}(r_y)\), and \(r_x \cap r_y = \emptyset\).

Where \(\text{set}(S)\) means \(S\) a proper set (\(S(x) \leq 0 \forall x\)). Denote this by \(\text{linearCtxt}(V, H)\). Whenever \(\text{linearCtxt}(V, H)\) holds, there is at most one path from a variable on the stack \(V\) to any location in \(H\).

6. Implementation and Evaluation

Implementation We have implemented the novel cost semantics and the type system in Resource Aware ML (RaML). The implementation covers full RaML, including user-defined data types, higher-order functions, and polynomial potential functions. However, there is no destructive match for function closures and analyzing the heap-space usage of closures still amounts to counting allocations only. The main changes that where necessary have been in the rules for sharing and pattern matching as described earlier. We also needed to change some elaboration passes that were not cost preserving anymore with the GC cost model.

The garbage collection cost semantics is implemented as an alternative evaluation module inside RaML. As mentioned before, RaML leverages the syntax of OCaml programs. First, we take the OCaml type checked abstract syntax tree and perform a series of transformations. The evaluation modules operates on the resulting RaML syntax tree. In the gc evaluation module, \texttt{evaluate} has the following signature:

\[
\text{evaluate} : \text{'a, unit} \text{Expressions.expression} \to \text{int} \to ((\text{'a value} * \text{'a heap} * \text{Int.Set.t}) \text{option})
\]
Here, the second argument int specifies the size of the initial freelist. The result is a tuple of the return value, heap, and freelist, or None in case the freelist was not sufficient for the evaluation. Whereas the normal evaluation boxes every value (everything evaluates to a location), the gc module follows the cost semantics and only boxes data constructors. The rationale is that the size for other values can be computed statically and stack allocated. One difference between the cost semantics and its implementation is that while in the language presented here list is the only data type, our implementation supports user defined data types. The extension is straightforward except the treatment of the nil constructor, or generally “empty” constructors that has arity zero. For simplicity of presentation, we evaluate all nil constructors to the same null value in the cost semantics. This is natural for lists because all nil constructors are the same, and every list has at most one nil node. However, for custom data types that have more than one kind of empty constructor, it is not possible to map every constructor to the same null value. Thus, the implementation treats all constructors uniformly, so each nil constructor also cost one heap location.

As mentioned before, all functions used in a program are declared in a global mutually recursive block, and we do not account for the constant space overhead for this block in the cost semantics. In order to implement this global function block, we allow closure creation during program evaluation. However, we allocate all closures from a separate freelist into a separate heap. This ensures that data constructors are allocated from the correct freelist and no space overhead is created by allocating closures for function declarations.

**Evaluation** We evaluated our new analysis on a number of functions. Table 1 contains a representative compilation. It shows the type signature for each function. Table 2 presents the test data that showcase the difference between the heap metric, the old analysis which only counts heap allocations, and the gc metric, which includes deallocations and copying cost for sharing. For each metric, we show the heap space bound computed by RaML, the number of constraints generated, and the time elapsed during analysis. The last column gives the expression for the exact heap high watermark derived by hand and verified by running the cost semantics.

Except for bfs and dfs, all functions in the table take a principal argument of type list. The variables in the table refer to this argument (for example, the type of the principal argument of quicksort is 'a list). In general, M refers to the number of cons constructors of the principal argument (or the number of outer cons nodes in case of nested lists); L refers to the maximum number of cons nodes of the inner lists.

For the sorting functions aside from mergesort, the new analysis using the gc metric derived asymptotically better bounds when compared to the heap metric. Furthermore, all bounds are exact with respect to the cost semantics. In regards to mergesort, the analysis was not able to derive a tight bound due to the limitations of AARA in deriving logarithmic bounds. A particularly nice result is that we can deduce for quicksort, the space usage is exactly 0, which justifies its use as a zero space-overhead sorting algorithm.

Next, we have the graph search algorithms operating on a binary tree. Again, the gc metric was able to derive exact space overheads, while the heap metric derived linear bounds for both. For transpose, the gc metric derived an asymptotically better bound, but was not able to derive the exact overhead. We implement matrices as lists-of-lists in row-major order. The transpose function is implemented tail-recursively, with the accumulator starting as the empty list. When “flipping” the first row r of the input and appending this to the accumulator, we need to create |r| many new nil and cons constructors to store the row as a column. While this overhead only occurs once, RaML is unable to infer this from the source code, and thus the cost is repeated over the entire input matrix, resulting in the linear bound (w.r.t the size of the matrix).

The last two functions demonstrate how the gc metric performs when there is variable sharing. map_it maps the input function across each list in the principal argument twice, returning a tuple of nested lists. The gc metric dictates that every outer data constructor in the principal argument needs to be copied, and thus gives the linear bound M + 1. In this case, the bound is exact. The pairs function takes a list and outputs a all pairs of the input list which are ordered ascending in input position. For example, pairs

### Table 1: Signature of Test Functions

<table>
<thead>
<tr>
<th>function</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>quicksort</td>
<td>['a -&gt; 'a -&gt; bool; 'a list] -&gt; 'a list</td>
</tr>
<tr>
<td>mergesort</td>
<td>[[‘a; ‘a] -&gt; bool; ‘a list] -&gt; 'a list</td>
</tr>
<tr>
<td>ocamlsort</td>
<td>[[‘a; ‘a] -&gt; bool; ‘a list] -&gt; 'a list</td>
</tr>
<tr>
<td>selection sort</td>
<td>int list -&gt; int list</td>
</tr>
<tr>
<td>eratosthenes</td>
<td>int list -&gt; int list</td>
</tr>
<tr>
<td>dfs</td>
<td>[btree; int] -&gt; btree option</td>
</tr>
<tr>
<td>bfs</td>
<td>[btree; int] -&gt; btree option</td>
</tr>
<tr>
<td>transpose</td>
<td>'a list list -&gt; 'a list list</td>
</tr>
<tr>
<td>map_it</td>
<td>['a -&gt; 'b; 'a list list] -&gt; 'b list list * 'b list list</td>
</tr>
<tr>
<td>pairs</td>
<td>'a list -&gt; (’a * ’a) list</td>
</tr>
</tbody>
</table>

Here, the second argument int specifies the size of the initial freelist. The result is a tuple of the return value, heap, and freelist, or None in case the freelist was not sufficient for the evaluation. Whereas the normal evaluation boxes every value (everything evaluates to a location), the gc module follows the cost semantics and only boxes data constructors. The rationale is that the size for other values can be computed statically and stack allocated. One difference between the cost semantics and its implementation is that while in the language presented here list is the only data type, our implementation supports user defined data types. The extension is straightforward except the treatment of the nil constructor, or generally “empty” constructors that has arity zero. For simplicity of presentation, we evaluate all nil constructors to the same null value in the cost semantics. This is natural for lists because all nil constructors are the same, and every list has at most one nil node. However, for custom data types that have more than one kind of empty constructor, it is not possible to map every constructor to the same null value. Thus, the implementation treats all constructors uniformly, so each nil constructor also cost one heap location.

As mentioned before, all functions used in a program are declared in a global mutually recursive block, and we do not account for the constant space overhead for this block in the cost semantics. In order to implement this global function block, we allow closure creation during program evaluation. However, we allocate all closures from a separate freelist into a separate heap. This ensures that data constructors are allocated from the correct freelist and no space overhead is created by allocating closures for function declarations.

**Evaluation** We evaluated our new analysis on a number of functions. Table 1 contains a representative compilation. It shows the type signature for each function. Table 2 presents the test data that showcase the difference between the heap metric, the old analysis which only counts heap allocations, and the gc metric, which includes deallocations and copying cost for sharing. For each metric, we show the heap space bound computed by RaML, the number of constraints generated, and the time elapsed during analysis. The last column gives the expression for the exact heap high watermark derived by hand and verified by running the cost semantics.

Except for bfs and dfs, all functions in the table take a principal argument of type list. The variables in the table refer to this argument (for example, the type of the principal argument of quicksort is 'a list). In general, M refers to the number of cons constructors of the principal argument (or the number of outer cons nodes in case of nested lists); L refers to the maximum number of cons nodes of the inner lists.

For the sorting functions aside from mergesort, the new analysis using the gc metric derived asymptotically better bounds when compared to the heap metric. Furthermore, all bounds are exact with respect to the cost semantics. In regards to mergesort, the analysis was not able to derive a tight bound due to the limitations of AARA in deriving logarithmic bounds. A particularly nice result is that we can deduce for quicksort, the space usage is exactly 0, which justifies its use as a zero space-overhead sorting algorithm.

Next, we have the graph search algorithms operating on a binary tree. Again, the gc metric was able to derive exact space overheads, while the heap metric derived linear bounds for both. For transpose, the gc metric derived an asymptotically better bound, but was not able to derive the exact overhead. We implement matrices as lists-of-lists in row-major order. The transpose function is implemented tail-recursively, with the accumulator starting as the empty list. When “flipping” the first row r of the input and appending this to the accumulator, we need to create |r| many new nil and cons constructors to store the row as a column. While this overhead only occurs once, RaML is unable to infer this from the source code, and thus the cost is repeated over the entire input matrix, resulting in the linear bound (w.r.t the size of the matrix).

The last two functions demonstrate how the gc metric performs when there is variable sharing. map_it maps the input function across each list in the principal argument twice, returning a tuple of nested lists. The gc metric dictates that every outer data constructor in the principal argument needs to be copied, and thus gives the linear bound M + 1. In this case, the bound is exact. The pairs function takes a list and outputs a all pairs of the input list which are ordered ascending in input position. For example, pairs
Table 2: Automatic Bound Analysis with RaML

<table>
<thead>
<tr>
<th>function</th>
<th>heap metric</th>
<th>gc metric</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>computed bound</td>
<td>constraints</td>
<td>time</td>
<td>computed bound</td>
<td>constraints</td>
<td>time</td>
<td>optimal</td>
<td></td>
</tr>
<tr>
<td>quicksort</td>
<td>1.00 + 3.50M + 1.50M²</td>
<td>8515</td>
<td>0.52</td>
<td>0</td>
<td>8519</td>
<td>0.48</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>mergesort</td>
<td>1.00 – 4.67M + 6.33M²</td>
<td>9572</td>
<td>0.64</td>
<td>−0.50M + 0.50M²</td>
<td>9578</td>
<td>0.58</td>
<td>$</td>
<td>\log(M)</td>
</tr>
<tr>
<td>ocamlsort</td>
<td>7.50 + 5.50M + 1.00M²</td>
<td>8565</td>
<td>0.51</td>
<td>1.00 + 1.00M</td>
<td>8573</td>
<td>0.50</td>
<td>$M + 1$</td>
<td></td>
</tr>
<tr>
<td>selection sort</td>
<td>2.00 + 3.00M + 1.00M²</td>
<td>639</td>
<td>0.06</td>
<td>0</td>
<td>642</td>
<td>0.05</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>eratosthenes</td>
<td>1.00 + 1.50M + 0.50M²</td>
<td>515</td>
<td>0.06</td>
<td>0</td>
<td>517</td>
<td>0.04</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dfs</td>
<td>3.00 + 2.00M</td>
<td>5481</td>
<td>0.90</td>
<td>2</td>
<td>5483</td>
<td>0.36</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>bfs</td>
<td>5.00 + 10.00M</td>
<td>24737</td>
<td>4.15</td>
<td>4</td>
<td>24742</td>
<td>1.62</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>transpose</td>
<td>1.00 + 3.50LM + 0.50LM²</td>
<td>10680</td>
<td>0.50</td>
<td>1.00 + 2.00LM</td>
<td>10684</td>
<td>0.50</td>
<td>$\max(0,2L – 1)$</td>
<td></td>
</tr>
<tr>
<td>map_it</td>
<td>2.00 + 2.00LM + 4.00M</td>
<td>30699</td>
<td>1.58</td>
<td>1.00M + 1.00</td>
<td>30703</td>
<td>1.57</td>
<td>$M + 1$</td>
<td></td>
</tr>
<tr>
<td>pairs</td>
<td>1.00 + 1.00M²</td>
<td>10214</td>
<td>0.60</td>
<td>0.50M + 0.50M²</td>
<td>10217</td>
<td>0.64</td>
<td>$0.5M² – 1.5M + 2$</td>
<td></td>
</tr>
</tbody>
</table>

In this article, we introduced a novel operational cost semantics that models a perfect tracing garbage collector and an extension to AARA that is sound with respect to the new semantics. We implemented the new semantics and analysis as modules in RaML and found through experimental testing that the extended AARA was able to derive asymptotically better bounds for several commonly used functions and programming patterns; often, the bounds are optimal with respect to the cost semantics.

Another direction for future work is using the cost-free metric $cf$ to model global garbage collection. In $cf$, all resource constants, including constructor nodes, are set to 0. A cost-free typing judgment then captures how an expression manipulates the structures in the context into the structure induced by its type. Using this fact, we could express maximum in the sequential composition \begin{align*}
\text{let} (e_1; x : \tau.e_2) \text{ by analyzing } e_1 \text{ twice—once with the cost-free metric and once with the regular metric—and assign potential to } x \text{ using the result type in the cost-free typing. In [23], the authors have successfully employed this cost-free metric to analyze parallel programs. Here, the difficulty is showing the simultaneous soundness of both destructive pattern matching and the cost-free composition. Another complication is the choice between local variable sharing and global context sharing. We leave the exploration of this area to future work.}
\end{align*}

Another direction for future work are function closures. The current treatment in our implementation is unsatisfactory since there is no equivalent to the destructive pattern match for closures. As a result, the GC metric in RaML only accounts for allocation of closures, which is not an improvement over the existing implementation. Ideally, we would like to account for deallocation at function applications and treat closures similar to other data structures in sharing. However, the size of closures cannot be determined easily statically and closures can currently not capture potential and are shared freely in RaML. As a result, the techniques we developed here do not directly carry over to closures.

Finally, we are interesting in exploring if our work can be used to improve the efficiency of garbage collection in languages like OCaml. A guaranteed upper bound on the heap space can be used in different ways to control the frequency of the collections and the total memory that is requested from the operating system.

[1;2;3;4] = [(1,2);(1,3);(1,4);(2,3);(2,4);(3,4)]. For pairs, the gc metric derived a bound that is asymptotically the same as the heap metric, but with better constants. An exact bound could not be derived because the deallocation potential from the pattern match in the definition of pairs is wasted because the matched body could already be typed with zero cost. However, this deallocation is used as usual in the cost semantics. Thus the slack in the bound totals to the size of the input.
References

A Notation

For a finite mapping \( f : A \to B \), we write \( \text{dom} \) for the defined values of \( f \). Sometimes we shorten \( x \in \text{dom}(f) \) to \( x \in f \). We write \( f[x \mapsto y] \) for the extension of \( f \) where \( x \) is mapped to \( y \), with the constraint that \( x \notin \text{dom}(f) \).

Given possibly non-disjoint sets \( A, B \), let the disjoint union be \( A \oplus B \) defined by \( \{\text{inl}, a \mid a \in A\} \cup \{\text{inr}, b \mid b \in B\} \).

Let a multiset be a function \( S : A \to \mathbb{N} \), i.e. a map of the multiplicity of each element in the domain.
Write $x \in S$ iff $S(x) \geq 1$. If for all $s \in S$, $\mu(s) = 1$, then $S$ is a property set, and we denote this by $\text{set}(S)$. Additionally, $A \cup B$ denotes counting union of sets where $(A \cup B)(s) = A(s) + B(s)$, similarly, $(A \cap B)(s) = \min A(s), B(s)$. Furthermore, $A \cup B$ denotes the usual union where $(A \cup B)(s) = \max (A(s), B(s))$. For the union of disjoint multi-sets $A$ and $B$, we write $A \cup B$ to emphasize the disjointness. For a collection of pairwise disjoint multi-sets $C$, i.e. $\forall X, Y \in C$. $X \cap Y = \emptyset$, we write $\text{disjoint}(C)$.

In the rest of the paper, we sometimes treat a set $A$ sets as multiset $A : A \rightarrow \mathbb{N}$ via $x \mapsto \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{o.w.} \end{cases}$ when convenient. For instance, if an operation defined on multisets is used on sets and multisets, the set is thus promoted.

Given a set $A$, let $\mathcal{P}(A)$ be the powerset of $A$. Given a multiset $A$, let $\varphi(A)$ be the power multiset of $A$, i.e. the set of all submultisets of $A$.

For a partition $f : A \rightarrow \mathcal{P}(B)$, we write the set of equivalence classes as $ec(f) = \{f(x) \mid x \in A\} = f(A)$, i.e. the image of $f$ on its domain $A$. Furthermore, a partition is proper if for any $x \in A$, $f(x) \neq \emptyset$.

Given a proper partition $f : A \rightarrow \mathcal{P}(B)$, for every $a \in A$, we can choose an arbitrary $b \in f(a)$ to be the representative for that part; call this $\text{rep}(a)$.

### B Linearity of Copy Semantics

In the soundness proof of $\text{FO}^{gc}$, we used an important lemma: that $E_{\text{copy}}$ is semantically linear, i.e. locations are used linearly. To see why, consider the second premise in the rule $\text{L:MatLD}$. In addition to the $p$ units of potential justified by the definition of linear potential, we get 1 unit from deallocating the cons cell itself. This is only sound if in the corresponding rule in $E_{\text{copy}}$, a location was actually collected. Consider the evaluation in question:

$$
V(x) = l \quad H(l) = \langle v_h, v_t \rangle \quad V'' = (V[x_h \mapsto v_h, x_t \mapsto v_t]) \mid_{FV(e_2)}
$$

$$
g = \{ l \in H \mid l \notin F \cup R \cup \text{locs}_{V''} \cup H(e_2) \} \quad V'', H, R, F \vdash g \downarrow v, v', F'
$$

If all the variables in $V$ was mapped to values with disjoint reachable sets, then we see that $l$ is only in the reachable set of $x$ (assuming that well-typed expressions don’t have duplicate occurrences of variables, i.e. $x \notin FV(e_1) \cup FV(e_2)$). Then it follows that $l \in g$ given that locations in $V$, $R$, and $F$ are also all disjoint, and this is what we needed to justify the rule $\text{L:MatL}$. Thus we have to show that $E_{\text{copy}}$ preserves the linearity invariant: given a linear computation, the evaluation result is also linear.

First, we characterize semantically linear contexts:

**Definition 8. (Linear context)** Given a context $(V, H)$, let $x, y \in \text{dom}(V)$, $x \neq y$, and $r_x = \text{reach}_H(V(x))$, $r_y = \text{reach}_H(V(y))$. It is linear given that:

1. $\text{set}(r_x), \text{set}(r_y)$
2. $r_x \cap r_y = \emptyset$

Denote this by $\text{linearCtx}(V, H)$.

Whenever $\text{linearCtx}(V, H)$ holds, visually, one can think of the stack as a collection of disjoint, directed trees with locations as nodes; consequently, there is at most one path from a variable on the stack $V$ to any location in $H$. Now we can formalize our intuition for linear computations:

**Definition 9 (Linear computation).** Given a configuration $C = (V, H, R, F)$ and an expression $e$, we say the 5-tuple $(C, e)$ is a computation; it is a linear computation given the following:

1. $\text{dom}(V) = FV(e)$
2. $\text{linearCtx}(V, H)$
3. disjoint(\{R, F, locs_{V,H}(e)\})

And we write linearComp(V, H, R, F, e) to denote this fact.

Given a semantically linear computation (one with no sharing between the underlying locations), the resulting value is linear (expressed by item 1. and 2. below):

Lemma 5 (Linearity of E\textsubscript{copy}). For all stacks V and heaps H, let V, H, R, F \vdash e \downarrow v, H', F' and \Sigma; \Gamma \vdash e : B. Then given that linearComp(V, H, R, F, e), we have the following:

1. set(reach\textsubscript{H'}(v))
2. disjoint(\{R, F', reach\textsubscript{H'}(v)\}), and
3. stable(R, H, H')

Where stable is a predicate on P(Loc) × Heap × Heap, defined below. The premises of this lemma is a subset of the premises of the soundness theorem. Thus, we could have merged the proof of this lemma directly into the soundness proof. However, we think makes the presentation clearer; furthermore, the linearity of E\textsubscript{copy} is interesting in itself, regardless of the accompanying type system. Some auxiliary lemmas:

Definition 10 (Stability). Given heaps H, H', a set of locations is stable if \forall l \in R. H(l) = H'(l). Denote this by stable(R, H, H').

Define \dagger : L^p(A) \mapsto L(A) as the map that erases resource annotations. This gives a simplified judgment \Sigma; \Gamma \vdash e : B^\dagger used in proofs where the resource annotations are not necessary.

Lemma 6. If \Sigma; \Gamma \dagger e : B, then \Sigma; \Gamma \vdash e : B^\dagger.

Proof. Induction on the typing judgement. □

Define \(FV^* : \text{Exp} \to \wp(\text{Var})\), the multiset of free variables of expressions, as the usual FV inductively over the structure of e. This version of FV reflects the multiplicity of variable occurrences.

Lemma 7. If \Sigma; \Gamma \dagger e : B, then set(FV^*(e)) and dom(\Gamma) = FV(e).

Proof. Induction on the typing judgement. □

Lemma 8. Let H \vdash v \mapsto a : A. For all sets of locations R, if reach\textsubscript{H}(v) \subseteq R and stable(R, H, H'), then H' \vdash v \mapsto a : A and reach\textsubscript{H'}(v) = reach\textsubscript{H'}(v).

Proof. Induction on the structure of H \vdash v \mapsto a : A. □

Corollary 9. Let H \vdash V : \Gamma. For all sets of locations R, if \bigcup_{x \in V} reach\textsubscript{H}(V(x)) \subseteq R and stable(R, H, H'), then H' \vdash V : \Gamma.

Proof. Follows from Lemma 8. □

Lemma 10 (stability of copying). Let H', v' = copy(H, L, v). For all l \in H, if l \notin L, then H(l) = H'(l). Further, reach\textsubscript{H'}(v') \subseteq L.

Lemma 11 (copy is copy). Let H', v' = copy(H, L, v). If H \vdash v \mapsto a : A, then H' \vdash v' \mapsto a : A.

Lemma 12. Let A \gamma^n A_1, A_2, H \vdash v_1 : A_1, H \vdash v_2 : A_2, and H \vdash v : A. Then \(\Phi_H(v : A) = \Phi_H(v_1 : A_1) + \Phi_H(v_2 : A_2) + n \cdot \lvert \text{dom}(\text{reach}_H(v)) \rvert\)

Lemma 13. Let H \vdash V : \Gamma, \Sigma; \Gamma \vdash e : A, and V, H \vdash e \downarrow v, H'. Then H' \vdash v : A.

Now the proof for linearity of E\textsubscript{copy}:
Proof. Nested induction on the evaluation judgement and the typing judgement.

Case 1: E:Var

Suppose \( H \models V : \Gamma, \text{dom}(V) = FV(e), \text{linearCtx}(V, H), \text{disjoint}(\{R, F, \text{locs}_{V,H}(e)\}) \)

\[
\begin{align*}
\text{set}(\text{reach}_H(v)) & \quad \text{(linearCtx}(V, H)) \\
\text{disjoint}(\{R, F, \text{reach}_H(v)\}) & \quad \text{(disjoint}(R, F)) \\
\text{linearCtx}(V, H) & \quad \text{(Sp.)} \\
\text{stable}(R, H, H') & \quad (H = H')
\end{align*}
\]

Case 2: E:Const* Due to similarity, we show only for E:ConstI

Suppose \( H \models V : \Gamma, \text{dom}(V) = FV(e), \text{linearCtx}(V, H), \text{disjoint}(\{R, F, \text{locs}_{V,H}(e)\}) \)

\[
\begin{align*}
\text{set}(\text{reach}_H(v)) & \quad (\text{reach}_H(v) = \emptyset) \\
\text{disjoint}(\{R, F, \emptyset\}) & \quad \text{(disjoint}(R, F)) \\
\text{linearCtx}(V, H) & \quad \text{(Sp.)} \\
\text{stable}(R, H, H') & \quad (H = H')
\end{align*}
\]

Case 4: E:App

Case 5: E:CondT Similar to E:MatNil

Case 6: E:CondF Similar to E:CondT

Case 7: E:Let

\[
\begin{align*}
V, H, R, F \vdash \text{let}(e_1; x : \tau.e_2) \downarrow v_2, H_2, F_2 & \quad \text{(case)} \\
V, H, R', F \vdash e_1 \downarrow v_1, H_1, F_1 & \quad \text{(ad.)} \\
\Sigma; \Gamma_1, \Gamma_2 \vdash \text{let}(e_1; x : \tau.e_2) : B & \quad \text{(case)} \\
\Sigma; \Gamma_1 \vdash e_1 : A & \quad \text{(ad.)}
\end{align*}
\]

Suppose \( H \models V : \Gamma, \text{dom}(V) = FV(e), \text{linearCtx}(V, H), \text{disjoint}(\{R, F, \text{locs}_{V,H}(e)\}) \)

\[
H \models V_1 : \Gamma_1 \quad \text{(def of W.D.E and Lemma 7)}
\]

By IH, we have invariant on the first premise

\[
\begin{align*}
(1) & \quad \text{dom}(V_1) = FV(e_1) \quad \text{(def of } V_1) \\
(2) & \quad \text{linearCtx}(V_1, H) \quad \text{(linearCtx}(V, H) \text{ and } V_1 \subseteq V) \\
(3) & \quad \text{disjoint}(R', F, \text{locs}_{V,H}(e_1))
\end{align*}
\]

\[
\begin{align*}
F \cap R' = \emptyset & \quad \text{(} F \cap \text{locs}_{V,H}(e) = \emptyset \text{ and } \text{locs}_{V_2,H}(\text{lam}(x : \tau.e_2)) \subseteq \text{locs}_{V,H}(e)\text{)} \\
F \cap \text{locs}_{V,H}(e_1) = \emptyset & \quad \text{(Lemma 7)} \\
\text{locs}_{V,H}(e_1) \cap \text{locs}_{V_2,H}(\text{lam}(x : \tau.e_2)) = \emptyset & \quad \text{(linearCtx}(V, H)) \\
R' \cap \text{locs}_{V,H}(e_1) = \emptyset & \quad \text{(disjoint}(\{R, \text{locs}_{V,H}(e)\})) \\
F \cap \text{locs}_{V,H}(e_1) = \emptyset & \quad \text{(Sp.)}
\end{align*}
\]

Thus we have disjoint(\( R', F, \text{locs}_{V,H}(e_1) \))

By IH,

\[
1.(1) \quad \text{set}(\text{reach}_{H_1}(v_1))
\]

\[
1.(2) \quad \text{disjoint}(\{R', F_1, \text{reach}_{H_1}(v_1)\})
\]
1.-(3) \( \text{stable}(R', H, H_1) \)
\[ V'_2, H_1, R, F_1 \cup g \vdash e_2 \downarrow v_2, H_2, F_2 \] (ad.)
\[ \Sigma, \Gamma_2, x : A \vdash e_2 : B \] (ad.)
\[ H_1 \models V'_2 : (\Gamma_2, x : A) \] (Lemma 13)

By IH, we have invariant on the second premise

NTS (1) - (3) to instantiate invariant on the second premise

(1) \( \text{dom}(V'_2) = FV(e_2) \) (def of \( V'_2 \))
(2) \( \text{linearCtxt}(V'_2, H_1) \)
Let \( x_1, x_2 \in V'2, x_1 \neq x_2 \) be arb.

\textbf{case:} \( x_1 \neq x, x_2 \neq x \)

\[ \text{reach}_H(V'_2(x_1)) \subseteq R' \] (def of \( V'_2 \))
\[ \text{reach}_H(V'_2(x_2)) \subseteq R' \] (same as above)
\[ \text{set}(\text{reach}_H(v_1)) \] (IH 1.1)
\[ \text{reach}_H(V'_2(x_2)) = \text{reach}_H(V(x_2)) \] (same as above)
\[ \text{set}(\text{reach}_H(V'_2(x_2))) \] (linearCtxt\((V, H))
\[ \text{reach}_H(V'_2(x_1)) \cap \text{reach}_H(V'_2(x_2)) = \emptyset \] (disjoint\((\{R', \text{reach}_H(v_1)\})\))

Thus we have \( \text{linearCtxt}(V'_2, H_1) \)

\textbf{(3)} \( \text{disjoint}(\{R, F_1 \cup g, \text{locs}_{V'_2, H_1}(e_2)\}) \)
\[ R \cap F_1 = \emptyset \] (disjoint\((\{R', F_1\})\) from 1.2 and \( R \subseteq R' \))
\[ R \cap (F_1 \cup g) = \emptyset \] (def of \( g \))
\[ \text{NTS} \ (F_1 \cup g) \cap \text{locs}_{V'_2, H_1}(e_2) = \emptyset \]

Let \( l \in \text{locs}_{V'_2, H_1}(e_2) \) be arb.
\[ l \in \text{reach}_H(V'_2(x')) \text{ for some } x' \in V'_2 \]

\textbf{case:} \( x' \neq x \)

\[ \text{reach}_H(V'_2(x')) = \text{reach}_H(V'_2(x')) \] (same as above)
\[ \text{reach}_H(V'_2(x')) \subseteq R' \] (def of \( R' \))
\[ \text{reach}_H(V'_2(x')) \cap F_1 = \emptyset \] (disjoint\((\{R', F_1\})\) from 1.2)

\textbf{case:} \( x' = x \)

\[ \text{reach}_H(V'_2(x')) = \text{reach}_H(v_1) \] (def of \( V'_2 \))
\[ \text{reach}_H(V'_2(x')) \cap F_1 = \emptyset \] (disjoint\((\{F_1, \text{reach}_H(v_1)\})\) from 1.2)
\[ \text{reach}_H(V'_2(x')) \subseteq \text{locs}_{V'_2, H_1}(e_2) \] (def of \( \text{locs}_{V, H} \))
\[ \text{reach}_H(V'_2(x')) \cap g = \emptyset \] (def of \( g \))

Thus \( \text{reach}_H(V'_2(x')) \cap (F_1 \cup g) = \emptyset \)
Case 11: E:Cons

Similar to E:Const

Case 9: E:MatP

Similar to E:MatCons

Case 8: E:Pair

Similar to E:Var

⊢ V,H,R,F

H

Suppose reach stable set (1)

NTS (1) - (3) holds after evaluation

reach (2)

disjoint set (1)

Lastly, showing (1) - (3) holds for the original case:

(1) set (reachH2 (v2)) (By 2.1)

(2) disjoint {R, F2, reachH2 (v2)} (By 2.2)

(3) stable (R, H1, H2)

Let l ∈ R be ar.

H (l) = H1 (l) (stable (R′, H, H1) from 1.3)

H1 (l) = H2 (l) (stable (R, H1, H2) from 2.3)

H (l) = H2 (l)

Hence stable (R, H, H2)

Case 8: E:Pair:

Similar to E:Var

Case 9: E:MatP:

Similar to E:MatCons

Case 10: E:Nil:

Similar to E:Const*

Case 11: E:Cons

V, H, R, F ⊨ e ⊏ l, H′′, F′

Suppose H ⊨ V : Γ, dom(V) = FV(e), linearCtx(V, H), disjoint {R, F, locsV,H(e)}

NTS (1) - (3) holds after evaluation

(1) set (reachH′′ (l))

stable {locsV,H(e)}, H, H′′ (disjoint {F, locsV,H(e)}) and copy only updates l ∈ L ⊆ F

reachH(V (x1)) = reachH′′(V (x1)) (reachH(V (xi)) ⊆ locsV,H(e) and 8 for i = 1, 2)

reachH′′(l) = {l} ∪ reachH′′(V (x1)) ∪ reachH′′(V (x2)) (def of reachH)

set (reachH′′ (l)) (l ∉ locsV,H(e) and linearCtx(V, H))

(2) disjoint {R, F′, reachH′′ (l)}


Case 12: E:MatNil

Suppose \( H \vdash V : \Gamma, \text{dom}(V) = FV(e), \text{linearCtxt}(V, H), \text{disjoint}(\{ R, F, \text{locsv}_{V,H}(e) \}) \)

\[
\begin{align*}
\Sigma; \Gamma' \vdash e_1 : B & \quad \text{ad.} \\
V, H, R, F \cup g \vdash e_1 \nvdash v, H', F' & \quad \text{ad.} \\
H \vdash V' : \Gamma' & \quad \text{def of W.D.E}
\end{align*}
\]

By IH, we have invariant on the premise

NTS (1) - (3) to instantiate invariant on the premise

1. \( \text{dom}(V') = FV(e_1) \) (def of \( V' \))
2. \( \text{linearCtxt}(V', H) \) (linearCtxt(\( V, H \)) and \( V' \subseteq V \))
3. \( \text{disjoint}(\{ R, F, \text{locsv}_{V',H}(e_1) \}) \) (disjoint(\( \{ R, F, \text{locsv}_{V,H}(e) \} \)) and \( \text{locsv}_{V',H}(e_1) \subseteq \text{locsv}_{V,H}(e) \))

Instantiating invariant on premise,

1. \( \text{set}(\text{reach}_{H'}(v)) \)
2. \( \text{disjoint}(\{ R, F_1, \text{reach}_{H'}(v) \}) \)
3. \( \text{stable}(R, H, H') \)

Case 13: E:MatCons

Suppose \( H \vdash V : \Gamma, \text{dom}(V) = FV(e), \text{linearCtxt}(V, H), \text{disjoint}(\{ R, F, \text{locsv}_{V,H}(e) \}) \)

\[
\begin{align*}
\Sigma; \Gamma' \vdash x : L(A) & \quad \text{ad.} \\
\Sigma; \Gamma', x_h : A, x_t : L(A) \vdash e_2 : B & \quad \text{ad.} \\
V'', H, R, F \cup g \vdash e_2 \nvdash v_2, H_2, F' & \quad \text{ad.}
\end{align*}
\]

Suppose \( H \vdash V : \Gamma, \text{dom}(V) = FV(e), \text{linearCtxt}(V, H), \text{disjoint}(\{ R, F, \text{locsv}_{V,H}(e) \}) \)

\[
\begin{align*}
H \vdash V(x) : L(A) & \quad \text{def of W.D.E} \\
H'' \vdash v_h : A, H'' \vdash v_t : L(A) & \quad \text{ad.} \\
H \equiv v_h : A, H \equiv v_t : L(A) & \quad \text{ad.} \\
H \vdash V'' : \Gamma', x_h : A, x_t : L(A) & \quad \text{def of W.D.E}
\end{align*}
\]

By IH, we have invariant on the premise

NTS (1) - (3) to instantiate invariant on the premise

1. \( \text{dom}(V'') = FV(e_2) \) (def of \( V'' \))
2. \( \text{linearCtxt}(V'', H) \)

Let \( x_1, x_2 \in V'', x_1 \neq x_2, r_{x_1} = \text{reach}_H(V''(x_1)), r_{x_2} = \text{reach}_H(V''(x_2)) \)

case: \( x_1 \notin \{ x_h, x_t \}, x_2 \notin \{ x_h, x_t \} \)

(1), (2) from \( \text{linearCtxt}(V, H) \)

case: \( x_1 = x_h, x_2 \notin \{ x_h, x_t \} \)

(1), (2) from \( \text{linearCtxt}(V, H) \)

\( \text{set}(r_{x_1}) \) (since \( \text{set}(\text{reach}_H(V(x))) \) from \( \text{linearCtxt}(V, H) \))
set(r_{x_2}) \quad \text{(since linearCtx(V, H))}
\]
x_2 \in FV(e) \quad \text{(def of FV)}
\]
reach_H(V(x)) \cap r_{x_2} = \emptyset \quad \text{(def of reach and linearCtx(V, H))}
\]
hence r_{x_1} \cap r_{x_2} = \emptyset
\]
\text{case: } x_1 = x_b, x_2 = x_t
\]
set(r_{x_1}) \text{ since set}(reach_H(V(x))) \text{ from linearCtx(V, H)}
\]
set(r_{x_2}) \text{ since set}(reach_H(V(x))) \text{ from linearCtx(V, H)}
\]
r_{x_1} \cap r_{x_2} = \emptyset \quad \text{(set}(reach_H(V(x)))
\]
\text{case: otherwise}
\]
similar to the above
\]
Thus we have linearCtx(V'', H)
\]
\text{(3) disjoint}([R, F \cup g, locs_{V''}, H(e_2)]) \quad \text{(def of g)}
\]
NTS R \cap locs_{V''}, H(e_2) = \emptyset
\]
Let l' \in locs_{V''}, H(e_2) be arb.
\]
\text{case: } l' \in reach_H(V''(x')) \text{ for some } x' \in FV(e_2) \text{ where } x' \notin \{x_b, x_t\}
\]
x' \in V \quad \text{(def of V'')}
\]
l' \in reach_H(V(x')) \quad \text{(def of reach)}
\]
x' \in FV(e) \quad \text{(def of FV)}
\]
l' \in locs_{V,H}(e) \quad \text{(def of locs_{V,H})}
\]
l' \notin R \quad \text{(disjoint}([R, F, locs_{V,H}(e)])
\]
\text{case: } l' \in reach_H(V''(x_b))
\]
tom l' \in reach_H(v_h)
\]
l' \in reach_H(V(x)) \quad \text{(def of reach)}
\]
l' \in locs_{V,H}(e) \quad \text{(def of locs_{V,H})}
\]
l' \notin R \quad \text{(since disjoint}([F, R, locs_{V,H}(e)])
\]
\text{case: } l' \in reach_H(V''(x_t))
\]
similar to above
\]
Hence R \cap locs_{V''}, H(e_2) = \emptyset \quad \text{(Similar to above)}
\]
F \cap locs_{V''}, H(e_2) = \emptyset \quad \text{(def. of g)}
\]
g \cap locs_{V''}, H(e_2) = \emptyset \quad \text{(Similar to above)}
\]
(F \cup g) \cap locs_{V'', H}(e_2) = \emptyset
\]
Thus disjoint([R, F \cup g, locs_{V''}, H(e_2)])
\]
Instantiating invariant on the premise,
\]
\text{(1) set}(reach_H(v))
\]
\text{(2) disjoint}([R, F', reach_H(v)])
\]
\text{(3) stable}(R, H, H')
\]

Case 13: E:Share
\]
e = \text{share } x \text{ as } x_1, x_2 \text{ in } e' \quad \text{(case)}
\]
Suppose H \models V : \Gamma, dom(V) = FV(e), linearCtx(V, H), disjoint([R, F, locs_{V,H}(e)]) \quad \text{(def. of wfc)
Let $V_2 = (V[x_1 \mapsto v', x_2 \mapsto v'']) \mid_{FV(e')}$

We show the subsequent computation is also well-formed to invoke the IH:

1. $\text{dom}(V_2) = \text{FV}(e')$  
   \hspace{1em} (dom(V) = FV(e) and def of FV)

2. $\text{linearCtxt}((V[x_1 \mapsto v', x_2 \mapsto v'']) \mid_{FV(e')}, H)$
   \hspace{1em} (definition of copy)

   Let $x' \mapsto v'' \in V'[x_1 \mapsto v']$. STS $\text{reach}_{H'}(v'') \cap \text{reach}_{H'}(v'') = \emptyset$

   \hspace{1em} (By Lemma 10)

   But since $F \cap \text{locs}_{V,H}(e) = \emptyset$, we have the result.  

3. $\text{disjoint}([R,F,\text{locs}_{V,H}(e)])$
   \hspace{1em} (linearComp(V,H,R,F,e))

   Disjointedness involving $F$ follows from assumption. Last one follows since $\text{locs}_{V',H'}(e') \subseteq \text{locs}_{V,H}(e) \cup L$

By IH:

1. $\text{set}(\text{reach}_{H''}(v))$

2. $\text{disjoint}([R,F',\text{reach}_{H''}(v)])$

3. $\text{stable}(R,H',H'')$

STS stable($R,H,H''$), which follows from $L \cap R = \emptyset$ and Lemma 10

\[\Box\]

\section{Soundness}

\textbf{Theorem 14 (Soundness).} let $H_o \models V_o : \Gamma, \Sigma; \Gamma \mid_{q^\prime} e : B$, $V_o, H_o \vdash e \downarrow v_o, H'_o$. Then $\forall C \in \mathbb{Q}^+$ and configuration $V,H,R,F$ s.t.

1. $V_o, H_o \sim V, H$

2. $\text{dom}(V) = FV(e)$

3. $\text{linearCtxt}(V,H)$

4. $\text{disjoint}([R,F,\text{locs}_{V,H}(e)])$

5. $|F| \geq \Phi_{V,H}(\Gamma) + q + C$

then there exists a triple $(v,H',F')$, and a freelist $F'$ s.t.

1. $V,H,R,F \vdash e \downarrow v, H', F'$

2. $v_o \sim_{H'} v$

3. $|F'| \geq \Phi_{H'}(v : B) + q' + C$

Call this the existence clause.

\textbf{Proof.} Nested induction on the evaluation judgement and the typing judgement.

\textbf{Case 1: E:Var}

\begin{align*}
V_o, H_o & \vdash x \downarrow V_o(x), H_o \\
\Sigma; x : B & \mid^t_{H} x : B
\end{align*}

\hspace{1em} (case)
Let $C \in \mathbb{Q}^+$, well-formed configuration $V, H, R, F$ s.t. $V_o, H_o \sim V, H$ and $|F| \geq \Phi_{V,H}(x : B) + q + C$

NTS the conclusions for the existence clause:

(1) $V, H, R, F \vdash e \Downarrow V(x), H, F$  
(E:Var)

(2) $V_o(x) \sim_{H_o} V(x)$  
(assumption)

(3) And we have $F \geq \Phi_{V,H}(x : B) + q + C$

$= \Phi_H(V(x) : B) + q + C$  
(definition of $\Phi$)

Case 2: E:Const*

Similar to E:Var

Case 4: E:App

$V_o, H_o \vdash f(x) \Downarrow v_o, H'_o$

$V_o(x) = v'_o$  
(admissibility)

Let $P(f) = (y_f, e_f)$

$[y_f \mapsto v'_o], H_o \vdash e_f \Downarrow v_o, H'_o$  
(admissibility)

$\Sigma; x : A \xrightarrow{\frac{q}{q}} f(x) : B$  
(case)

$\Sigma(f) = A \xrightarrow{\frac{q}{q}} f : B$  
(admissibility)

Let $V(x) = v', g = \text{collect}(R, \text{locs}_{V,H}(e_f), H, F)$

By IH on $[y_f \mapsto v'], H, R, F \cup g$, have the existence clause. NTS the following conditions:

(1) $[y_f \mapsto v_o'], H_o \sim [y_f \mapsto v'], H$  
($v'_o \sim_{H_o} v'$ by assumption)

(2) - (4) Have by assumption

(5) NTS $|F \cup g| \geq \Phi_{[y_f \mapsto v'], H}(x : A) + q + C$

$\Sigma; y_f : A \xrightarrow{\frac{q}{q}} e_f : B$  
(def of W.F.E)

Applying the existence clause and E:App, we're done.

Case 5: E:CondT

$\Gamma = \Gamma', x : \text{bool}$  
(ad.)

$H \vdash V : \Gamma'$  
(def of W.F.E)

$\Sigma; \Gamma' \xrightarrow{\frac{q}{q}} e_t : B$  
(ad.)

$V, H, R, F \cup g \vdash e_t \Downarrow v, H', F'$  
(ad.)

$|F \cup g| - |F'| \leq \Phi_{V,H}(\Gamma) + q - (\Phi_{H'}(v : B) + q')$  
(IH)

$|F| - |F'| \leq \Phi_{V,H}(\Gamma) + q - (\Phi_{H}(v : B) + q')$

Case 6: E:CondF

Similar to E:CondT
Case 7: E: Let

\[ V_\omega, H_\omega \vdash e \downarrow v_\omega''', H_\omega''' \] (case)

\[ V_\omega' = V_\omega \mid_{\text{dom}(FV(e_1))} \] (ad.)

\[ V_\omega'', H_\omega' \vdash e_1 \downarrow v_\omega'', H_\omega'' \] (ad.)

\[ \Sigma; \Gamma \models \frac{q}{q} \text{ let}(e_1; x : \tau.e_2) : B \] (case)

\[ \Gamma = \Gamma_1, \Gamma_2 \] (ad.)

\[ \Sigma; \Gamma_1 \models \frac{q}{p} e_1 : A \] (ad.)

\[ H \models V_\omega' : \Gamma_1 \] (def of \( \models \))

Let \( C \in \mathbb{Q}^+ \), well-formed configuration \( V, H, F, F \) s.t. \( V_\omega, H_\omega \sim V, H \) and \( |F| \geq \Phi_{V,H}(\Gamma) + q + C \)

NTF \( v_2, H_2, F_2 \) s.t.

1. \( V, H, F \vdash e \downarrow v_2, H_2, F_2 \) and

2. \( v_2'' \sim_{H_2} v_2 \)

3. \( |F_2| \geq \Phi_{H_2}(v_2 : B) + q + C \)

Let \( R' = R \cup \text{locs}_{V,H}(FV(e_2) \setminus \{x\}) \)

\[ \text{disjoint}([R', F, \text{locs}_{V,H}(e_1)]) \] (Similar to case in Lemma 5)

Let \( V_1 = V \mid_{\text{dom}(FV(e_1))} \)

Instantiate IH with \( C = C + \Phi_{V_2,H_2}(\Gamma_2), V = V_1, H = H, F = F, R = R' \)

we get existence clause on the first premise

NTS (1) - (4) to instantiate existence clause on the first premise:

1. \( V_\omega'', H_\omega' \sim V_1, H \) (assumption)

2. (4) Same as in 5

3. \( |F| \geq \Phi_{V_1,H}(\Gamma_1) + q + C \)

\( \Phi_{V,H}(\Gamma) \geq \Phi_{V_1,H}(\Gamma_1) + \Phi_{V,H}(\Gamma_2) \) (Similar to case in Lemma 5)

Instantiating existence clause on the first premise, we get \( v_1, H_1, F_1 \) s.t.

Fact 1. \( V_1, H, R', F \vdash e_1 \downarrow v_1, H_1, F_1 \)

Fact 2. \( v_\omega' \sim_{H_1} v_1 \)

Fact 3. \( |F_1| \geq \Phi_{H_1}(v_1 : A) + p + C + \Phi_{V_2,H_1}(\Gamma_2) \)

For the second premise:

\[ V_\omega'' = (V_\omega[x \mapsto v_\omega']) \mid_{FV(e_2)} \] (ad.)

\[ V_\omega'', H_\omega' \vdash e_2 \downarrow v_\omega'', H_\omega'' \] (ad.)

\[ \Sigma; \Gamma_2, x : A \models \frac{p}{q} e_2 : B \] (ad.)

\[ H_1 \models v_\omega' : A \] (By Lemma 8 and 13)

\[ H_1 \models V_2 : \Gamma_2, x : A \] (def of \( \models \))

Let \( V_2 = (V[x \mapsto v_1]) \mid_{FV(e_2)} \)

Let \( g = \{ l \in H_1 \mid l \notin F_1 \cup R \cup \text{locs}_{V_2,H_1}(e_2) \} \)

Instantiate IH with \( C = C, V = V_2, H = H_1, F = F_1 \cup g, R = R \), we get existence clause on the second premise

NTS (1) - (4) to instantiate existence clause on the second premise:

1. \( V_\omega''', H_\omega' \sim V_2, H_1 \) (assumption and Fact 2.)

2. (4) Same as Lemma 5

3. \( |F_1 \cup g| \geq \Phi_{V_2,H_1}(\Gamma_2, x : A) + p + C \)
Case 8: E:Pair Similar to E:Const*

Case 9: E:MatP Similar to E:MatCons

Case 10: E:Nil Similar to E:Const*

Case 11: E:Cons

\[
V_o, H_o \vdash \text{cons}(x_1; x_2) \Downarrow l_o, H'_o \quad (\text{case})
\]

\[
H'_o(l_o) = \langle V_o(x_1), V_o(x_2) \rangle \quad (\text{admissibility})
\]

\[
\Sigma; x_1 : A, x_2 : L^p(A) \vdash \frac{q + p + 1}{q} \text{cons}(x_1; x_2) : L^p(A)
\]

Let C \in \mathbb{Q}^+, well-formed configuration V, H, R, F s.t.

\[
V_o, H_o \sim V, H \quad \text{and} \quad |F| \geq \Phi_{C,H}(x_1 : A, x_2 : L^p(A)) + (p + q + 1) + C
\]

Let v = ∥V(x_1), V(x_2)∥, l \in F, H' = H\{l \mapsto v\}, F' = F \setminus \{l\}

(1) V, H, R, F \vdash \text{cons}(x_1; x_2) \Downarrow l, H', F'

(E:Cons)

(2) l_o \sim_{H'} l

(3) |F'| \geq \Phi_{H'}(l : L^p(A)) + q + C

Let E:Cons

Case 12: E:MatNil Similar to E:Cond*

Case 13: E:MatCons

\[
V_o(x) = l_o \quad (\text{ad.})
\]

\[
H_o(l_o) = \langle v'_o, v''_o \rangle \quad (\text{ad.})
\]

\[
\Gamma = \Gamma', x : L^p(A) \quad (\text{ad.})
\]

\[
\Sigma; \Gamma', x_h : A, x_t : L^p(A) \vdash \frac{q + p + 1}{q} e_2 \quad (\text{ad.})
\]

\[
V'_o = \langle V_o[x_h \mapsto v'_o, x_t \mapsto v''_o] \rangle |_{FV(e_2)} \quad (\text{ad.})
\]

\[
V'_o, H_o \vdash e_2 \Downarrow v_o, H' \quad (\text{ad.})
\]

\[
H \vdash V(x) : L^p(A) \quad (\text{def of \vdash})
\]

\[
H \vdash v_h : A, H \vdash v_t : L^p(A) \quad (\text{Inversion on \vdash})
\]

\[
H \vdash V'_o \Gamma', x_h : A, x_t : L^p(A) \quad (\text{def of W.D.E})
\]

Let C \in \mathbb{Q}^+, well-formed configuration V, H, R, F s.t. V_o, H_o \sim V, H and |F| \geq \Phi_{V,H}(\Gamma) + q + C

Let NTF v, H', F' s.t.

1. \[ V, H, R, F \vdash e \Downarrow v, H', F' \] and

2. \[ v_o \sim_{H'} v \]

2. |F'| \geq \Phi_{H'}(v : B) + q' + C
Let $V(x) = l$

$H(l) = \langle v_h, v_t \rangle$ \hspace{100pt} (l_o \sim_H^o l$ by assumption)

Let $V' = (V[x_h \mapsto v_h, x_t \mapsto v_t])|_{FV(e_2)}$

Let $g = \{ l \in H \mid l \notin F \cup R \cup locs_{V', H}(e_2) \}$

NTS $g$ nonempty, in particular, that $l \in g$

$l \notin F \cup R$ \hspace{100pt} (l \in locs_{V', H}(e_2) and disjoint(\{R, locs_{V', H}(e)\}))

AFSOC $l \in locs_{V', H}(e_2)$

Then $l \in reach_H(V'(x'))$ for some $x' \in dom(V')$

**case** $x' \in \{x_h, x_t\}$:

WLOG let $x' = x_h$

But then $reach_H(V(x))(l) \geq 2$ and $set(reach(V(x)))$ doesn't hold

**case** $x' \notin \{x_h, x_t\}$:

$x' \in dom(V)$

$x = x'$ \hspace{100pt} (l \in reach_H(V(x)) and linearCtxt(V, H))

Contradiction, $x \notin dom(v(V'))$

$l \notin locs_{V', H}(e_2)$

Hence $l \in g$

By IH with $C' = C, V = V', H = H, R = R, F_1 = F \cup g$ we have the existence clause. NTS the following:

(1) $V'_0, H_o \sim V', H$ \hspace{100pt} (assumption)

(2) - (4) Same as Lemma 5

(5) $|F_1| \geq \Phi_{V', H}(\Gamma', x_h : A, x_t : L^p(A)) + q + p + 1 + C$

$= |F| + |g|$ \hspace{100pt} (F and g disjoint)

$\geq \Phi_{V', H}(\Gamma) + q + C + |g|$ \hspace{100pt} (assumption)

$= \Phi_{V', H}(\Gamma', x_h : A, x_t : L^p(A)) + q + p + 1 + C$ \hspace{100pt} (definition of $\Phi$)

(5) $|F'| \geq \Phi_{H'}(v : B) + q' + C$

By existence clause have $v, H', F'$ s.t.

Fact 1. $V', H, R, F \cup g \vdash e_2 \Downarrow v, H', F'$

Fact 2. $|F'| \geq \Phi_{H'}(v : B) + q' + C$

Apply E:MatCons and we’re done

**Case 14: E:Share**

$V_o, H_o \vdash \text{share } x \text{ as } x_1, x_2 \text{ in } c' \Downarrow v_o, H'_o$ \hspace{100pt} (case)

$V_o(x) = v'_o$ \hspace{100pt} (ad.)

$V'_o = (V_o[x_1 \mapsto v'_o, x_2 \mapsto v'_o])|_{FV(c')}$

$V'_o, H_o \vdash c' \Downarrow v_o, H'_o$ \hspace{100pt} (ad.)

$\Sigma; \Gamma, x : A \vdash \frac{\text{share } x \text{ as } x_1, x_2 \text{ in } c'}{v_o} B$ \hspace{100pt} (case)

$A \land A_1, A_2, 1$ \hspace{100pt} (ad.)

$\Sigma; \Gamma, x_1 : A_1, x_2 : A_2 \vdash \frac{\text{share } x \text{ as } x_1, x_2 \text{ in } c'}{v_o} B$ \hspace{100pt} (ad.)

Let $C \in \mathbb{Q}^+$, well-formed configuration $V, H, R, F$ s.t. $V_o, H_o \sim V, H$ and $|F| \geq \Phi_{V, H}(\Gamma, x : A) + q + C$

NTF $v, H'', F''$ s.t.
1. \(V, H, R, F \vdash e \downarrow v, H'', F''\) and

2. \(x_o \sim^{H''}_{H''} v\)

3. \(|F''| \geq \Phi_{H''}(v : B) + q' + C\)

Let \(V(x) = v', L \subseteq F, |L| = |\text{dom}(\text{reach}_H(v'))|, H', v'' = \text{copy}(H, L, v'), V' = (V[x_1 \mapsto v', x_2 \mapsto v'']) |_{F_V(v)}\)

\[F' = F' \setminus L, g = \{l \in H \mid l \notin F' \cup R \cup \text{locs}_{V', H'}(e)\}\]

By IH with \(C, V', H', R, F', g\), have the existence clause. NTS the following:

(1) \(V'_o, H_o \sim V', H'\) (By Lemma ?? and assumption)

(2) - (4) same as lemma 5

(5) \(|F' \cup g| \geq \Phi_{V', H'}(\Gamma, x_1 : A_1, x_2 : A_2) + q + C\)

\(\text{STS} \left| (F \setminus L) \cup g \right| \geq \Phi_{V', H'}(\Gamma, x_1 : A_1, x_2 : A_2) + q + C\)

\[\iff \left| (F - |L|) + |g| \geq \Phi_{V', H'}(\Gamma) + \Phi_{V', H'}(x_1 : A_1) + \Phi_{V', H'}(x_2 : A_2) + q + C\right.\]

\(\text{STS} \left| F \right| \geq \Phi_{V', H'}(\Gamma) + \Phi_{V', H'}(x : A) + q + C\) (By Lemma 12)

\(\iff \left| F \right| \geq \Phi_{V', H'}(\Gamma, x : A) + q + C\) (By Lemma 10)

Instantiate the existence clause and apply E:Share, and we’re done

\[\square\]

## D \(E_{\text{copy}}\) over-approximates \(E_{\text{gc}}\)

**Definition 11.** A configuration \((V, H, R, F)\) is well-formed if

1. \(\text{dom}(H) \subseteq \text{reach}_H(V) \cup R \cup F\)

2. \(\text{reach}_H(V) \cup R \subseteq \text{dom}(H) \setminus F\)

3. \(\text{collect}(R, \text{reach}_H(V), H, F) = \emptyset\)

For a context \((V, H, R, F)\), denote the garbage w.r.t a set of locations \(L\) as \(\text{collect}(R, L, H', F') = \{l \in H' \mid l \notin F' \cup R \cup L\}\).

**Lemma 15.** Given a well-formed context \((V, H, R, F)\) and \(V, H, R, F \vdash e \downarrow v, H', F', \text{dom}(H') \subseteq \text{reach}_H(v) \cup R \cup F', \text{reach}_H(v) \cup R \subseteq \text{dom}(H') \setminus F'\) and \(\text{collect}(R, \text{reach}_H(v), H', F') = \emptyset\).

Now consider two well-formed configurations \(C_1 = (V_1, H_1, R_1, F_1), C_2 = (V_2, H_2, R_2, F_2)\).

A mapping \(f : A \rightarrow \mathcal{P}(B)\) is a **partition** on \(B\) the image of \(A\) forms a disjoint union of \(B\) (e.g. \(\forall x, y \in A, f(x) \cap f(y) = \emptyset \land \bigcup f(A) = B\)). Furthermore, a partition is **proper** if for any \(x \in A\), \(f(x) \neq \emptyset\).

Given a proper partition \(f\), we can choose an arbitrary \(b \in f(a)\) to be the representation for that part; call this singlet set \(\{b\} \text{ rep}(a)\).

A simple corollary is the fact that if \(V_2, H_2\) is a linear context (e.g. \(\text{linearCtx}(V_2, H_2)\) holds), then \(|\gamma(l)| = |(\text{reach}_{H_1}(V_1))(l)|\), where \(\text{reach}_{H_1}(V_1) = \bigcup_{x \in \text{dom}(V_1)} \text{reach}_{H_1}(x)\). In general for a multiset \(S\), when this holds, we say that \(\gamma\) is a **counting partition** for \(S\).

For a partition \(f : A \rightarrow \mathcal{P}(B)\), we write the set of equivalence classes as \(\text{cc}(f) = \{f(x) \mid x \in A\} = f(A)\), i.e. the image of \(f\) on its domain \(A\).
Lemma 17. Let $\text{dir}$ be the set $\{L,R,N\}$, denoting left, right, and next respectively. We can index values via directions:

$$
get_H(\text{Just}((v_1,v_2),L)) = \text{Just}(v_1)$$
$$
get_H(\text{Just}((v_1,v_2),R)) = \text{Just}(v_2)$$
$$
get_H(\text{Just}((v_1,v_2)),r) = \text{None}$$
$$
get_H(\text{Just}(l),N) = \text{Just}(H(l))$$
$$
get_H(\text{Just}(l),L) = \text{None}$$
$$
get_H(r,\_\_\_) = r$$

Let $P$ be a sequence of directions. Extend $\text{get}$ to sequence of directions:

$$\text{find}_H(v,D::P) = \text{find}_H(\text{get}_H(v,D),P)$$
$$\text{find}_H(v,[\_]) = v$$

Call $P$ valid w.r.t a value $v$ if $\text{find}_H(v,P) = \text{Just}(v')$ for some $v'$. Write $V_H(x,P)$ for $\text{fromJust}(\text{find}_H(V(x),P))$ given a valid sequence $P$ w.r.t $V(x)$, and $\text{reach}_H(V(x),P)$ for $\text{reach}_H(V_H(x,P))$. Given a map $m : X \rightarrow S(\text{dir})$ from variables to valid sequences of directions, Define $\text{reachPath}_{V,H}(X,m) = \bigcup_{x \in X} \text{reach}_H(V(x;m(x)))$.

Lemma 16. Let $V_1, H_1 ~ V_2, H_2$. Then for all $x \in \text{dom}(V_1)$ and sequence of directions $P$, Either $\text{find}_H(V_1(x),P) = \text{find}_H(V_2(x),P) = \text{None}$ or $\text{find}_H(V_1(x),P) = v_1$, $\text{find}_H(V_2(x),P) = v_2$ and $v_1 \sim_{H_2} v_2$.

Proof. Induction on length of $P$ and then $H \vdash v \rightarrow a : A$. \qed

Definition 13. A configuration $C_2 = (V_2, H_2, R_2, F_2)$ is a copy extension of another configuration $C_1 = (V_1, H_1, R_1, F_1)$ iff

1. $V_1, H_1 ~ V_2, H_2$
2. There is a proper partition $\gamma : \text{dom}(H_1) \setminus F_1 \rightarrow \mathcal{P}(\text{dom}(H_2) \setminus F_2)$ such that for all $l \in \text{dom}(\gamma)$, $|\gamma(l)| = \text{reach}_H(V_1)(l) + R_1(l)$
3. for all $l \in \text{dom}(\gamma)$, $x \in \text{dom}(V_1)$, valid sequence of directions $P$ w.r.t $V_1(x)$, $|\text{reach}_H(V_2(x),P) \cap \gamma(l)| = \text{reach}_H(V_1(x,P))(l)$.
4. for all $l \in \text{dom}(\gamma)$, $|\gamma(l) \cap R_2| = R_1(l)$
5. $|F_1| = |F_2| + |\otimes(\gamma)|$, where $\otimes(\gamma) = \bigcup_{P \in \text{ext}(\gamma)} C \setminus \text{rep}(C)$

Write this as $C_1 \preceq C_2$.

Note that $\preceq$ is reflexive. Now the key lemma:

Lemma 17. Let $(C_2,e)$ be a linear computation. Given that $C_2 \vdash_{\text{copy}} e \downarrow w, H', F'$, and $H' \vdash e \rightarrow a : A$, for all well-formed configurations $C_1$ such that $C_1 \preceq C_2$, there is exists a triple $(w,Y',M') \in \text{Val} \times \text{Heap} \times \text{Loc}$ and $\gamma : \text{dom}(Y') \setminus M' \rightarrow \mathcal{P}(\text{dom}(H') \setminus F')$ s.t.

1. $C_1 \vdash_{\text{free}} e \downarrow w, Y', M'$
2. $w \sim_{H'} Y', w$
3. $\gamma'$ is a proper partition, such that for all $l \in \text{dom}(\gamma)$, $|\gamma'(l)| = \text{reach}_{Y_1}(w_1)(l) + S(l)$
4. For all $P$ valid w.r.t $v$, $|\text{reach}_{Y'}(\text{find}_{H'}(v,P)) \cap \gamma'(l)| = \text{reach}_{Y'}(\text{find}_{Y'}(v,P))(l)$
5. $l \in \text{dom}(\gamma')$, $\gamma'(l) \cap R = \gamma(l) \cap R$
Lemma 18. Let $V_2, H_2, R_2, F_2 \vdash v, H', F'$, and $V_1, H_1, R_1, F_1 \leq V_2, H_2, R_2, F_2$ because \(-, \gamma, \eta, -, -\). Then the following hold: for all $l \in \dom(H_1) \setminus F_1$, $X \subseteq \dom(V)$ as \(\text{clean}(C, L) = \{l \in H \mid l \notin F \cup R \cup L\}\). Some auxiliary lemmas:

Proof. 

\[ \gamma(l) \cap (F_2 \cup R_2 \cup \text{reachPath}_{V_2, H_2}(X, m)) = \emptyset \]  

(definition of \text{clean})

\[ \text{NTS } l \notin F_1 \cup R_1 \cup \text{reach}_{H_1}(X) \]

(1) $l \notin F_1$  

(assumption)

(2) $l \notin R_1$

Know $\gamma(l) \cap R_2 = \emptyset$

$|\gamma(l) \cap R_2| = R_1(l) = 0$

(3) $l \notin \text{reachPath}_{V_1, H_1}(X, m)$

Thus, $l \in \text{clean}(C_1, \text{reachPath}_{V_1, H_1}(X, m))$

$\iff$

\[ l \notin (F_1 \cup R_1 \cup \text{reachPath}_{V_1, H_1}(X, m)) \]

\[ \text{NTS } \gamma(l) \cap (F_2 \cup R_2 \cup \text{reachPath}_{V_2, H_2}(X, m)) = \emptyset \]

(1) $\gamma(l) \cap F_2 = \emptyset$  

(assumption)

(2) $\gamma(l) \cap R_2 = \emptyset$

$|\gamma(l) \cap R_2| = R_1(l)$

$= 0$

(3) $\gamma(l) \cap \text{reachPath}_{V_2, H_2}(X, m) = \emptyset$

Let $x \in X$

$|\gamma(l) \cap \text{reach}_{H_2}(V_2(x; m(x)))| = \text{reach}_{H_1}(V_1(x; m(x)))(l)$

(3) of $\preceq$

$|\gamma(l) \cap \text{reach}_{H_2}(V_2(x; m(x)))| = 0$

$\gamma(l) \cap \text{reach}_{H_2}(V_2(x; m(x))) = \emptyset$

$\gamma(l) \cap \text{reachPath}_{V_2, H_2}(X, m) = \emptyset$

$\gamma(l) \cap (F_2 \cup R_2 \cup \text{reachPath}_{V_2, H_2}(X, m)) = \emptyset$

$\gamma(l) \in \text{clean}(C_2, \text{reachPath}_{V_2, H_2}(X, m))$

\[ \square \]
Lemma 19. Given a proper partition \( \gamma : \text{dom}(Y) \setminus M \rightarrow \mathcal{P}(\text{dom}(H) \setminus F) \), equivalent values \( v \sim_H w \), and the following:

1. \( L \subseteq F \)
2. \( |L| = |\text{dom}(\text{reach}_H(v))| \)
3. \( H', v' = \text{copy}(H, L, v) \)
4. \( F' = F \setminus L \)

there is a proper partition \( \gamma' : \text{dom}(Y) \setminus M \rightarrow \mathcal{P}(\text{dom}(H) \setminus F') \) s.t. \( |\gamma'(l)| = |\gamma(l)| + |\text{reach}_Y(w)(l)| \), \( |\text{reach}_H(v) \cap \gamma'(l)| = |\text{reach}_H(v') \cap \gamma(l)| \), \( |\text{reach}_H(v') \cap \gamma'(l)| = |\text{reach}_H(v) \cap \gamma(l)| \).

Proof. Induction on \( \text{copy} \).  

Now the proving \( \mathcal{E}_{\text{copy}} \) over approximates \( \mathcal{E}_{\text{gc}} \):

Proof. Induction on the evaluation judgement.

Case 1: \( \text{E:Var} \) Trivial

Case 2: \( \text{E:Const}^* \) Trivial

Case 4: \( \text{E:App} \) Similar to \( \text{E:CondT} \)

Case 5: \( \text{E:CondT} \)

\[ V, H, R, F \vdash_{\text{E}_{\text{copy}}} \text{if}(x; e_1; e_2) \Downarrow v, H', F' \quad \text{(case)} \]

Let \( W, Y, S, M \leq V, H, R, F \)

Let \( W' = W \upharpoonright_{\text{dom}(V)} \)

Let \( j = \{ l \in Y | l \notin M \cup S \cup \text{locs}_{W,Y}(e_1) \} \)

NTS \( W', Y, S, M \cup j \leq V', H, R, F \cup g \)

(1) \( W, Y \sim V, H \) \( \text{(W,} Y \sim V, H) \)

(2) NTF a proper partition \( \gamma' : \text{dom}(Y) \setminus (M \cup j) \rightarrow \mathcal{P}(\text{dom}(H) \setminus (F \cup g)) \)

Let \( \gamma'(l) = \gamma \upharpoonright_{\text{dom}(Y) \setminus (M \cup j)}(l) \setminus g \)

First, show \( \gamma' \) is a partition

Let \( l, l' \in \text{dom}(Y) \setminus (M \cup j) \)

\( \gamma'(l) \cap \gamma'(l') = \emptyset \) \( \text{(}\gamma \text{ is partition)} \)

\( \gamma'(\text{dom}(Y) \setminus (M \cup j)) = \gamma(\text{dom}(Y) \setminus (M \cup j)) \setminus g \)

\( = (\bigsqcup_{l \in \text{dom}(Y) \setminus M} \gamma(l) \setminus \bigsqcup_{l \in j} \gamma(l)) \setminus g \)

\( = ((\text{dom}(H) \setminus F) \setminus \bigsqcup_{l \in j} \gamma(l)) \setminus g \) \( \text{(}\gamma \text{ is partition)} \)

\( = (\text{dom}(H) \setminus F) \setminus g \)

\( = \text{dom}(H) \setminus (F \cup g) \)

Hence \( \gamma' \) is a partition

\( \gamma' \) is also proper:

Let \( l \in \text{dom}(Y) \setminus (M \cup j) \)

\( \gamma'(l) = \gamma(l) \setminus g \)
AFSOC $\gamma(l) \subseteq g$

$l \in j$ (By Lemma 18)

Contradiction since assumed $l \notin j$

Now, NTS $|\gamma'(l)| = reach_Y(W')(l) + S(l)$

Let $l \in \text{dom}(\gamma')$

$\gamma'(l) = \gamma(l) \setminus g$ (definition)

$|\gamma'(l)| = |\gamma(l)| - |\gamma(l) \cap g|$

$= reach_Y(W)(l) + S(l) - |\gamma(l) \cap g|$

$g = reach_Y(V |_{\text{dom}(V) \setminus \text{FV}(e_1)})$

$|\gamma(l) \cap g| = reach_Y(W |_{\text{dom}(W) \setminus \text{FV}(e_1)})(l)$ (condition 3. of $\preceq$)

Thus, $|\gamma'(l)| = reach_Y(W)(l) + S(l) - reach_Y(W |_{\text{dom}(W) \setminus \text{FV}(e_1)})(l)$

$= reach_Y(W |_{\text{dom}(e_1)})(l) + S(l)$

$= reach_Y(W')(l) + S(l)$

(3) Let $l \in \text{dom}(\gamma'), x \in \text{dom}(W'), P$ valid sequence w.r.t $W'(x)$. NTS

$|reach_H(V'(x; P)) \cap \gamma'(l)| = reach_Y(W'(x; P))(l)$

STS $\{reach_H(V'(x; P)) \cap (\gamma(l) \setminus g) = reach_Y(W'(x; P))(l)

$g \cap reach_H(V'(x; P)) = \emptyset$ (definition of $g$

$|reach_H(V'(x; P)) \cap (\gamma(l) \setminus g) = (reach_H(V'(x; P)) \setminus g) \cap \gamma(l)|$

$= |reach_H(V'(x; P)) \cap \gamma(l)|$

$= |reach_H(V(x; P)) \cap \gamma(l)|$

$= reach_Y(W(x; P))(l)$

$= reach_Y(W'(x; P))(l)$ (condition 3. of $\preceq$)

(4) $S \subseteq \text{dom}(Y) \setminus (M \cup j)$ since $S \cap j = \emptyset$

Let $l \in S$. NTS $|\gamma'(l) \cap R| = S(l)$

STS $|\gamma'(l) \cap g| = S(l)$

$(\gamma(l) \setminus g) \cap R = S(l)$

$S(l) = S(l)$ (condition 4. of $\preceq$

(5) NTS $|M \cup j| = |F \cup g| + |\otimes (\gamma')|$

STS $|M| + |j| = |F| + |g| + |\otimes (\gamma')|$

By assumption $|M| = |F| + |\otimes (\gamma)|$

STS $|j| + |\otimes (\gamma)| = |g| + |\otimes (\gamma')|$

NTF a bijection $f : j \oplus \otimes (\gamma) \rightarrow g \sqcup \otimes (\gamma')$

First, we know that $g = \bigsqcup_{l \in \mathcal{L}} \gamma(l) \sqcup L$ for some $L$

(By 18)

Let $\mathcal{C}_1 = \{\gamma(l) \mid l \in j\}; \mathcal{C}_2 = \text{ec}(\gamma) \setminus \mathcal{C}_1$

Clearly, $\otimes (\gamma) = \bigsqcup_{C \in \mathcal{C}_1} C \setminus \text{rep}(C) \sqcup \bigsqcup_{C \in \mathcal{C}_2} C \setminus \text{rep}(C)$

Let $D_1 = \bigsqcup_{C \in \mathcal{C}_1} C \setminus \text{rep}(C), D_2 = \bigsqcup_{C \in \mathcal{C}_2} C \setminus \text{rep}(C)$

We define the bijection $f$ by parts: $f_1 : j \oplus D_1 \rightarrow \bigsqcup_{C \in \mathcal{C}_2} C, f_2 : D_2 \rightarrow L \sqcup \otimes (\gamma')$
\( f_1(x) = \begin{cases} \text{rep}(\gamma(l)) & x = (\text{inl}, l) \\ l & x = (\text{inr}, l) \end{cases} \)

Clearly, \( f_1 \) is a bijection, and \(|j| + |D_1| = \bigcup_{C \in C_1} |C|\)

To avoid the problem of maintaining a single representative for a class (which might be collected), note the following:

\(|C_2| = |\text{ec}(\gamma) \setminus \{\gamma(l) \mid l \in j\}| = |\text{ec}(\gamma \setminus \text{dom}(Y) \setminus (M \cup j))| = |\text{ec}(\gamma')|\)

Meaning that \( C_2 \) has the same number of classes as \( \gamma' \) (note these classes might be different)

Since both \( \gamma, \gamma' \) are proper partitions we have the following:

\(|D_2| = |L \cup \bigoplus \gamma'\big| \iff \bigcup_{C \in C_2} C \setminus \text{rep}(C) = \bigcup_{C \in \text{ec}(\gamma')} C \setminus \text{rep}(C)\big| \iff \bigcup_{C \in C_2} |C| = |L \cup \bigoplus C|\)

In fact, the latter two sets are equal:

let \( l' \in \bigcup_{C \in C_2} C \)

\( l' \in H \setminus F \) \hspace{1cm} (Def. of partition)

**case** \( l \in g \)

\( l' \notin \bigcup_{l \in j} \gamma(l) \) \hspace{1cm} (Def. of \( C_2 \))

\( l' \in L \) \hspace{1cm} (Def. of \( g \))

\( l' \in L \cup \bigcup_{C \in \text{ec}(\gamma')} C \)

**case** \( l' \notin g \)

\( l' \in H \setminus (F \cup g) \)

Exists \( C \in \text{ec}(\gamma') \) s.t. \( l' \in C \) \hspace{1cm} (Def. of partition)

\( l' \in \bigcup_{C \in \text{ec}(\gamma')} C \)

\( l' \in L \cup \bigcup_{C \in \text{ec}(\gamma')} C \)

For the other direction, let \( l' \in L \cup \bigcup_{C \in \text{ec}(\gamma')} C \)

**case** \( l' \in L \)

\( l' \in H \setminus F \) \hspace{1cm} (Def. of \( L \))

Exists \( C \in \text{ec}(\gamma) \) s.t. \( l' \in C \) \hspace{1cm} (Def. of partition)

\( l' \in \bigcup_{C \in \text{ec}(\gamma)} C \)

**case** \( l' \in \bigcup_{C \in \text{ec}(\gamma')} C \)

\( l' \in H \setminus (F \cup g) \) \hspace{1cm} (Def. of partition)

\( l' \in H \setminus F \)
Case 7: E:Let

Similar to E:CondT

Case 6: E:CondF  Similar to E:CondT

Case 7: E:Let

\[ V, H, R, F \vdash \text{let}(e_1; x : \tau.e_2) \Downarrow v_2, H_2, F_2 \]  
\[ W, Y, S, M \preceq V, H, R, F \]  
\[ V_1, H, R', F \vdash \text{let}(e_1) \Downarrow v_1, H_1, F_1 \]  
Let \( W_1 = W \mid_{\text{FV}(e_1)}, S' = S \cup \text{locs}_W.Y(\text{lam}(x : \tau.e_2)) \) 

NTS \( W_1, Y, S', M \preceq V_1, H, R', F \)

(1) \( W_1, Y \sim V_1, H \)  

(2) \( \gamma \) is a proper partition  

(3) \( \gamma' \) is a proper partition, and \( |\gamma''(l)| = \text{reach}_Y.(w)(l) + R(l) \)  

(4) \( \gamma''(l) \cap R = \gamma'(l) \cap R \)  

(5) \( |M'| = |F'| + |\ominus(\gamma'')| \)  

Apply F:CondT to (1), we are done.

Case 6: E:CondF  Similar to E:CondT

Existence \( C \in ec(\gamma) \) s.t. \( l' \in C \)  

\[ \gamma' \]  

Hence we show that \( |D_2| = |L \cup \ominus(\gamma')| \), and together with the previous equality, \( |j| + |\ominus(\gamma)| = |g| + |\ominus(\gamma')| \)  

Thus we have \( W', Y, S, F \cup j \preceq V', H, R, F \cup g \)  

\[ V', H, R, F \cup g \vdash_{\text{copy}} e_1 \Downarrow v, H', F' \]  

By IH on \( (V', H, R, F \cup g) \), we have \( (w, Y', M', \gamma'') \) such that

1. \( W', Y, S, M \cup j \vdash_{\text{copy}} e_1 \Downarrow w, Y', M' \)
2. \( v \sim_{H'} w \)
3. \( \gamma' \) is a proper partition, and \( |\gamma''(l)| = \text{reach}_Y.(w)(l) + R(l) \)
4. \( \gamma''(l) \cap R = \gamma'(l) \cap R \)
5. \( |M'| = |F'| + |\ominus(\gamma'')| \)

Apply F:CondT to (1), we are done.
Let \( x' \in FV(e_2) \setminus \{x\} \)
\[|\gamma(l) \cap reach_H(V(x'))| = reach_Y(W(x'))(l)\] (condition 3. of \( \preceq \))

(5) Have \(|M| = |F| + |\circ (\gamma)|\) (condition 5. of \( \preceq \))

By IH on first premise, there is \((w_1, Y_1, M_1)\) and \(\gamma_1\) s.t.

Fact 1. \(W', Y, S', M \vdash^{e_2} e \Downarrow w_1, Y_1, M_1\)

Fact 2. \(v \sim_{Y_1} w\)

Fact 3. \(\gamma_1\) is a proper partition, such that for all \(l \in dom(\gamma_1)\),
\[|\gamma_1(l)| = |reach_{Y_2}(w_1)(l)| + S(l)\]

Fact 4. For all \(P, \{reach_{H_1}(find_{H_1}(w_1; P)) \cap \gamma_1(l)| = reach_{Y_1}(find_{Y_1}(w_1; P))(l)\)

Fact 5. \(\gamma_1(l) \cap R' = \gamma(l) \cap R'\) and \(S' \subseteq dom(Y_1) \setminus M_1\)

Fact 6. \(|M_1| = |F_1| + |\circ (\gamma_1)|\)

\(V_2, H_1, R, F_1 \cup g \vdash e_2 \Downarrow v_2, H_2, F_2\) (admissibility)

Let \(W_2 = \{W[x \mapsto w_1] \}_{FV(e_2)}\), \(j = \{l \in H_1 | l \notin M_1 \cup S \cup locs\}_{W_2, Y_1}(e_2)\)

NTS \(W_2, Y_1, S, M_1 \cup j \subseteq V_1, H_1, R, F_1 \cup g\)

(1) \(W_2, Y_1 \sim W_2, H_1\) (condition 1. of \( \preceq \) and Fact 1)

(2) Let \(\gamma_2 : dom(Y_1) \setminus (M_1 \cup j) \rightarrow \mathcal{P}(dom(H_1) \setminus (F_1 \cup g))\) be defined by \(\gamma_2(l) = \gamma_1(l) \setminus g\)

NTS \(|\gamma_2(l)| = reach_{Y_1}(W_2)(l) + S(l)\)
STS \(|\gamma_1(l)| = |\gamma_1(l) \cap g| + reach_{Y_1}(W_2)(l) + S(l)\)

case \(x \in FV(e_2)\):
\[
g = \text{collect}(R', v_1, H_1, F_1)
\]
\[
= 0
\]
\[
|\gamma_1(l)| = reach_{Y_1}(w_1)(l) + S'(l)
\]
\[
= reach_{Y_1}(w_1)(l) + reach_Y(W \upharpoonright_{FV(e_2) \setminus \{x\}})(l) + S(l)
\]
\[
= reach_{Y_1}(W_2)(l) + S(l)
\]

case \(x \notin FV(e_2)\):
\[
g = \text{collect}(R', v_1, H_1, F_1) \sqcup reach_{H_1}(v_1)
\]
\[
= reach_{H_1}(v_1)
\]
\[
|\gamma_1(l) \cap g| = reach_{Y_1}(w_1)
\]
STS \(|\gamma_1(l)| = reach_{Y_1}(w_1)(l) + reach_{Y_1}(W_2)(l) + S(l)\)
\[
= reach_{Y_1}(w_1)(l) + S'(l)
\]
\[
= reach_{Y_1}(w_1)(l) + reach_Y(W \upharpoonright_{FV(e_2) \setminus \{x\}})(l) + S(l)
\]
\[
= reach_{Y_1}(W_2)(l) + S(l)
\]
\[
= reach_{Y_1}(W_2)(l) + S(l)
\]
(3) Let \(l \in dom(\gamma_2), x' \in dom(W_2), P\) valid sequence w.r.t \(W_2(x')\). NTS
\[\{reach_{H_1}(V_2(x'; P)) \cap \gamma_2(l)| = reach_{Y_1}(W_2(x'; P))(l)\]

case \(x' = x\):
\[|\gamma_1(l) \cap \gamma_2(l)| = reach_{Y_1}(W_2(x'; P))(l)\] (Fact 4)

case \(x' \neq x\):
\[|\gamma_1(l) \cap \gamma_2(l)| = reach_{Y_1}(W_2(x'; P))(l)\] (stability and condition 3. of \( \preceq \))
\[|\gamma_1(l) \cap \gamma_1(l)| = reach_{Y_1}(W_2(x'; P))(l)\] (condition 5. of \( \preceq \))
\[|\gamma_1(l) \cap \gamma_2(l)| = reach_{Y_1}(W_2(x'; P))(l)\] (\(reach_{H_1}(V_2(x'; P)) \subseteq R'\) and Fact 5. \(g \cap R' = \emptyset\))
Case 8: E:Pair  Similar to E:Const*

Case 9: E:MatP  Similar to E:CondT

Case 10: E:Nil  Similar to E:Const*

Case 11: E:Cons

\[
(4).a \quad \text{NTS } S \subseteq \text{dom}(Y_1) \setminus (M_1 \cup j)
\]
\[
S' \subseteq \text{dom}(Y_1) \setminus M_1 \\
S \subseteq \text{dom}(Y_1) \setminus M_1 \\
S \cap j = \emptyset
\]
\[
S \subseteq \text{dom}(Y_1) \setminus (M_1 \cup j)
\]

(4).b  Let \( l \in S \). NTS \( |\gamma_2(l) \cap R| = S(l) \)
\[
|\gamma_2(l) \cap R| = |(\gamma_1(l) \setminus g) \cap R|
\]
\[
= |\gamma_1(l) \cap R|
\]
\[
= |\gamma(l) \cap R|
\]
\[
= S(l)
\]

(5)  NTS \( |M_1 \cup j| = |F_1 \cup g| + \circ(\gamma_2) \)

Exactly the same as in E:CondT

Thus \( W_2, Y_1, S, M_1 \cup j \preceq V_2, H_1, R, F_1 \cup g \)

By IH on the second premise, we have \((w_2, Y_2, M_2)\) and \(\gamma_3\) s.t.

Fact 1'.\( W_2, Y_1, S, M_1 \cup j \vdash F_2 \downarrow w_2, Y_2, M_2 \)

Fact 2'.\( \vdash Y_2 \downarrow w_2 \)

Fact 3'.\( \gamma_3 \) is a proper partition, such that for all \( l \in \text{dom}(\gamma_3) \),
\[
|\gamma_3(l)| = |\text{reach}_{Y_2}(w_2)(l)| + \gamma_3(l) \\
\]

Fact 4'.For all \( P, |\text{reach}_{H_2}(\text{find}_{H_2}(v_2; P)) \cap \gamma_3(l)| = \text{reach}_{Y_2}(\text{find}_{Y_2}(w_2; P))(l) \)

Fact 5'.\( \gamma_3(l) \cap R = \gamma_2(l) \cap R \) and \( S \subseteq \text{dom}(Y_2) \setminus M_2 \)
\[
\gamma_2(l) \cap R = \gamma_1(l) \cap R
\]
\[
= \gamma(l) \cap R
\]

(5).  NTS \( |M_2| = |F_2| + |\circ(\gamma_3)| \)

Apply F:Let to (1), we are done.
Let \( l \in \text{dom}(\gamma') \), \( P \) be a valid path w.r.t \( l \). NTS \(|\text{reach}_H((\text{find}_H; (l; P)) \cap \gamma'(l'))| = |\text{reach}_Y((\text{find}_Y; (m; P)))(l')|

\begin{align*}
\text{case } P = [] & \quad |\text{reach}_H((\text{find}_H; (l; P)) \cap \gamma'(l'))| = |\text{reach}_H((l) \cap \gamma'(l'))| \\
& \quad = |\{l\} \cup \text{reach}_H(v_1) \cup \text{reach}_H(v_2) \cap \gamma'(l')| \\
& \quad = |\{l\} \cap \gamma(l')| + |\text{reach}_H(v_1) \cap \gamma'(l')| + |\text{reach}_H(v_2) \cap \gamma'(l')| \\
& \quad = 1 \cdot m + \text{reach}_Y^r(w_1)(l') + \text{reach}_Y^r(w_2)(l') \\
& \quad = \text{reach}_Y^r((m)(l')) \\
& \quad = \text{reach}_Y^r((\text{find}_Y; (m; P)))(l') \\
\end{align*}

\begin{align*}
\text{case } P = N :: P' & \quad |\text{reach}_H((\text{find}_H; (l; N :: P')) \cap \gamma'(l'))| = |\text{reach}_H((v_1, v_2) \cap \gamma'(l'))| \\
& \quad = \text{reach}_Y^r((\text{find}_Y; (m; P'))(l') \\
\end{align*}

\begin{align*}
(5) \text{ NTS } S \subseteq \text{dom}(\gamma') & \quad \text{(condition 4. of } \leq \text{ and } \text{dom}(\gamma) \subseteq \text{dom}(\gamma')) \\
\text{Let } l' \in \text{dom}(\gamma'), \text{ NTS } \gamma(l') \cap R = \gamma(l) \cap R \\
\text{STS } \gamma[m \mapsto \{l\}](l') \cap R = \gamma(l') \cap R
\end{align*}

\begin{align*}
\text{case } l' = m : & \quad \gamma[m \mapsto \{l\}](m) \cap R = \{l\} \cap R \\
& \quad = \emptyset \\
& \quad \text{(well-formed configuration)}
\end{align*}

\begin{align*}
\text{case } l' \neq m : & \quad \gamma[m \mapsto \{l\}](m) \cap R = \gamma(l') \cap R \\
(6) \text{ NTS } |M'| = |F'| + \odot(\gamma') & \quad |M'| = |M| - 1 \\
& \quad |F'| = |F| - 1 \\
& \quad \odot(\gamma') = \odot(\gamma[m \mapsto \{l\}]) \\
& \quad = \bigcup C \in \text{ec}(\gamma') C \setminus (\text{rep}(C)) \\
& \quad = \{\{l\} \setminus \{l\}\} \cup \bigcup C \in \text{ec}(\gamma) C \setminus (\text{rep}(C)) \\
& \quad = \bigcup C \in \text{ec}(\gamma) C \setminus (\text{rep}(C)) \\
& \quad = \odot(\gamma) \\
\text{STS } |M| - 1 = |F| - 1 + \odot(\gamma) & \quad \text{(condition 5. of } \leq \text{)}
\end{align*}

Have \(|M| = |F| + \odot(\gamma)|
Case 12: E:MatNil  Similar to E:Cond*

Case 13: E:MatCons

\[ V, H, R, F \vdash \text{match } x \{ \text{nil} \mapsto e_1 \mid \text{cons}(x; x_1) \mapsto e_2 \} \Downarrow v, H', F' \]  
\text{(case)}

\[ W, Y, S, M \leq V, H, R, F \]  
\text{(assumption)}

\[ V', H, R, F \cup g \vdash e_2 \Downarrow v, H', F' \]  
\text{(admissibility)}

Let \( W' = W \lvert_{\text{dom}(V')} \)
Let \( j = \{ l \in Y \mid l \notin M \cup S \cup \text{locs}_{W', Y}(e_2) \} \)

\[ \text{NTS } W', Y, S, M \cup j \leq V', H, R, F \cup g \]

Let \( \gamma' : \text{dom}(Y) \setminus (M \cup j) \rightarrow \mathcal{P}(\text{dom}(H) \setminus (F \cup g)) \) be defined by \( \gamma'(l) = \gamma(l) \setminus g \)

\[ (1) \ W', Y \sim V', H \]  
\text{(similar to E:CondT)}

Let \( W(x) = m \)

\[ H(m) = (w_h, w_l), v_h \sim^H_{W} w_h, v_l \sim^H_{W} w_l \]  
\text{(1)}

\( (2) \) \( \gamma'(l) \) is a proper partition .  
\text{(similar to E:CondT)}

Let \( l' \in \text{dom}(\gamma') \)

Now, \( \text{NTS } |\gamma'(l')| = \text{reach}_Y(W')(l) + S(l') \)

Consider \( |\gamma(l') \cap \text{reach}_H(V(x))| \)

\[ = |\gamma(l') \cap \{ l \}| + |\gamma(l') \cap \text{reach}_H(v_h)| + |\gamma(l') \cap \text{reach}_H(v_l)| \]

\[ = \text{reach}_Y(W(x))(l') \]  
\text{(condition 3. of \( \leq \))}

\[ = \{ m \mapsto 1 \} \cup \text{reach}_Y(w_h) \cup \text{reach}_Y(w_l)(l') \]

\[ = \{ m \mapsto 1 \}(l') + \text{reach}_Y(w_h)(l') + \text{reach}_Y(w_l)(l') \]

Note \( |\gamma(l') \cap \text{reach}_H(v_h)| = |\gamma(l') \cap \text{reach}_H(V(x; [N, L]))| \)

\[ = \text{reach}_Y(W(x; [N, L]))(l') \]

\[ = \text{reach}_Y(w_h)(l') \]

Similarly, \( |\gamma(l') \cap \text{reach}_H(v_l)| = \text{reach}_Y(w_h)(l') \)

Thus \( |\gamma(l') \cap \{ l \}| = \{ m \mapsto 1 \}(l') = \mathbb{1}_{l' = m} \)

Back to the NTS:

\[ \gamma'(l') = |\gamma(l') \setminus g| \]  
\text{(definition)}

\[ |\gamma'(l')| = |\gamma(l')| - |\gamma(l') \cap g| \]

\[ = \text{reach}_Y(W')(l') + S(l') - |\gamma(l') \cap g| \]

\text{case } \{ x_h, x_t \} \subseteq \text{FV}(e_2) : \quad \text{(definition of } g \text{ and 5)}

\[ g = \{ l \} \cup \text{reach}_H(V \lvert_{\text{dom}(V) \setminus (\text{FV}(e_2) \cup \{ x \})}) \]

\[ |\gamma(l') \cap g| = (|\gamma(l') \cap \{ l \}| \cup (\gamma(l') \cap \text{reach}_H(V \lvert_{\text{dom}(V) \setminus (\text{FV}(e_2) \cup \{ x \})}))) \]  
\text{(condition 3. of \( \leq \))}

Thus, \( |\gamma(l')| = \text{reach}_Y(W')(l') + S(l') - (\mathbb{1}_{l' = m} + \text{reach}_Y(W \lvert_{\text{dom}(W) \setminus (\text{FV}(e_2) \cup \{ x \}))(l')}) \)

\[ = \text{reach}_Y(W \lvert_{\text{FV}(e_2)})(l') + S(l') - \mathbb{1}_{l' = m} \]

\[ = \text{reach}_Y(W \lvert_{\text{FV}(e_2)})(l') + \text{reach}_Y(m)(l') + \text{reach}_Y(w_h)(l') + \text{reach}_Y(w_l)(l') + S(l') - \mathbb{1}_{l' = m} \]

\[ = \text{reach}_Y(W \lvert_{\text{FV}(e_2)})(l') + \text{reach}_Y(w_h)(l') + \text{reach}_Y(w_l)(l') + S(l') \]

\[ = \text{reach}_Y(W')(l') + S(l') \]

\text{case } x_h \in \text{FV}(e_2), x_t \notin \text{FV}(e_2) : \quad \text{(definition of } g \text{ and 5)}

\[ g = \{ l \} \cup \text{reach}_H(V \lvert_{\text{dom}(V) \setminus (\text{FV}(e_2) \cup \{ x \})}) \cup \text{reach}_H(v_t) \]
|γ′(l′) ∩ g| = |γ′(l′) ∩ {l}| + |γ′(l′) ∩ reach_H(V | dom(V) \ (FV(ε) \cup \{x\}))| + |γ′(l′) ∩ reach_H(v)|

(\text{condition 3. of } ≤)

= 1_{l=m} + reach_H(V \mid \text{dom}(V) \setminus (FV(ε) \cup \{x\})) (l′) + reach_Y (w_i (l′))

(\text{condition 3. of } ≤)

Thus, |γ′(l′)| = reach_H(W) (l′) + S(l′) - (1_{l=m} + reach_H(V \mid \text{dom}(V) \setminus (FV(ε) \cup \{x\})) (l′) + reach_Y (w_i (l′))

= reach_H(W \mid FV(ε)) (l′) + reach_Y (m) (l′) + reach_Y (w_i (l′)) + reach_Y (w_i (l′)) + S(l′) - 1_{l=m} - reach_Y (w_i (l′))

= reach_Y (W \mid FV(ε)) (l′) + reach_Y (w_i (l′)) + S(l′)

= reach_H(W′) (l′) + S(l′)

case \ x_1 ∈ FV(ε), \ x_2 \notin FV(ε) \text{ and } \{x_h, x_1\} ∩ FV(ε) = \emptyset: \text{ symmetric to above}

(3) Let l′ ∈ \text{dom}(γ′), x′ ∈ \text{dom}(W′), P \text{ valid sequence w.r.t } W'(x'). \text{ NTS}

|reach_H(V′(x′; P)) \cap γ′(l′)| = reach_H(W′(x′; P)) (l′)

case x′ \notin \{x_h, x_1\}:

|reach_H(V′(x′; P)) \cap γ′(l′)| = |reach_H(V′(x′; P)) \cap (γ′(l′) \setminus g)|

= |reach_H(V′(x′; P)) \cap γ′(l′)|

= |reach_H(W′(x; P)) (l′)|

= |reach_H(W′(x; P)) (l′)|

= reach_H(W′(x; P)) (l′)

= \text{reach}_H(W′(x'; P)) (l′)

|reach_H(V′(x′; P)) \cap γ′(l′)| = |reach_H(f ind_H(v_h, P)) \cap γ′(l′)|

|reach_H(f ind_H(v_h, P)) \cap γ′(l′)|

= |reach_H(f ind_H(l, N :: L :: P)) \cap γ′(l′)|

= |reach_H(V′(x, N :: L :: P)) \cap γ′(l′)|

= reach_H(W′(x, N :: L :: P)) (l′)

= reach_H(W′(x, N :: L :: P)) (l′)

= reach_H(W′(x; P)) (l′)

= reach_H(W′(x; P)) (l′)

= \text{reach}_H(W′(x'; P)) (l′)

(4) Similar to \text{E:CondT}

(5) Similar to \text{E:CondT}

Apply IH and \text{F:MatCons} and we’re done

Case 14: \text{E:Share}

V, H, R, F \vdash \text{share } x \text{ as } x_1, x_2 \text{ in } e \downarrow v, H'', F′  \quad \text{(case)}

W, Y, S, M ≤ V, H, R, F  \quad \text{(assumption)}

V′, H′, R, F′ \cup g \vdash e \downarrow v, H'', F′  \quad \text{(admissibility)}

Let W(x) = w′

v ′ ∼_H w′  \quad \text{(condition 1 of } ≤)

Let W′ = (W[x_1 \mapsto w′, x_2 \mapsto w′] |_{FV(ε)})

Let j = \{ l ∈ Y | l \notin M \cup S \cup locs_{W′, Y}(e) \}

\text{NTS } W′, Y, S, M ▷ j ≤ V′, H, R, F \cup g

Let γ′ : dom(Y) \setminus (M ▷ j) → P(dom(H) \setminus (F \cup g)) be defined by γ′(l) = γ(l) \setminus g

(1) W′, Y ∼ V′, H  \quad \text{(Similar to E:CondT)}

(2) Have proper partition γ′ : dom(Y) \setminus M → dom(H′) \setminus F′  \quad \text{(By 19)}

Let j = \{ l ∈ H | l \notin M \cup S \cup locs_{W′, Y}(e) \}
Let $\gamma''(l) : \text{dom}(Y) \setminus (M \cup j) \to \text{dom}(H') \setminus (F' \cup g)$ by defined by $\gamma''(l) = \gamma'(l) \setminus g$

$\gamma''$ is a proper partition (Similar to E:CondT)

Let $l \in \text{dom}(\gamma'')$

Now, NTS $|\gamma''(l)| = \text{reach}_Y(W')(l) + S(l')$

$\gamma''(l) = \gamma'(l) \setminus g$ (definition)

$|\gamma''(l)| = |\gamma'(l)| - |\gamma'(l) \cap g|

= |\gamma(l)| + 1_{l \in \text{reach}_Y(w')} - |\gamma'(l) \cap g|

= \text{reach}_Y(W)(l) + S(l) + \text{reach}_Y(w')(l) - |\gamma'(l) \cap g|

\textbf{case} \ \{x_1, x_2\} \subseteq \text{FV}(e) :

$g = \text{collect}(R, \text{reach}_H(V), H, F) = \emptyset$ (well-formed configuration)

$|\gamma''(l)| = \text{reach}_Y(W)(l) + S(l) + \text{reach}_Y(w')(l)$ (By 19)

$= \text{reach}_Y(W')(l) + S(l)$

\textbf{case} $x_1 \in \text{FV}(e), x_2 \notin \text{FV}(e) :

g = \text{collect}(R, \text{reach}_H(V), H, F) \cup \text{reach}'_H(v') = \text{reach}_H'(v')$ (well-formed configuration)

$|\gamma''(l)| = \text{reach}_Y(W)(l) + S(l) + \text{reach}_Y(w')(l) - |\gamma'(l) \cap \text{reach}_H'(v')|$

$= \text{reach}_Y(W)(l) + S(l) + \text{reach}_Y(w')(l) - |\gamma(l) \cap \text{reach}_H(V)|$ (By 19)

$= \text{reach}_Y(W)(l) + S(l) + \text{reach}_Y(w')(l) - \text{reach}_Y(w')(l) + \text{reach}_Y(w')(l)$ (condition 3. of $\preceq$)

$= \text{reach}_Y(W)(l) + S(l)$

$= \text{reach}_Y(W')(l) + S(l)$

\textbf{case} $x_2 \in \text{FV}(e), x_1 \notin \text{FV}(e)$ and $\{x_1, x_2\} \cap \text{FV}(e) = \emptyset$: symmetric to above

$(3) - (5)$ Similar to E:MatCons

Applying the IH then F:Share, and we're done.